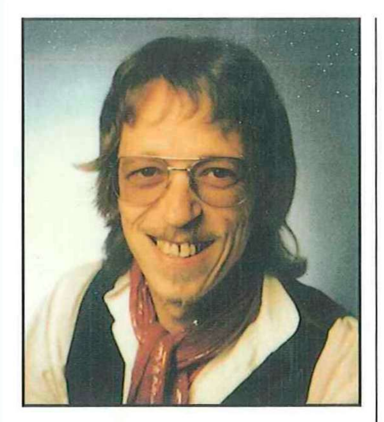


Die faszinierende Welt der Grafik erklärt an zahlreichen Anwendungsbeispielen in BASIC 7.0 und ASSEMBLER: Sprites ★ Shapes ★ Animation

GRAFIK-PROGRAMMIERUNG C128

Heimo Ponnath

Incl. Disk mit allen Programmen



HEIMO PONNATH
geboren am 15.9.1944 in Hamburg, ist Physikochemiker. Seine erste Berührung mit Informatik fand 1966 statt. Seitdem fasziniert ihn der scheinbare Gegensatz zwischen den strengen Regeln der Mathematik und den vielfältigen Möglichkeiten der Grafik. Der Heimcomputer als Brücke dazwischen und als Mittel zur Kreativität für jeden verführte ihn zum Schreiben. Heute ist Ponnath als freier Journalist tätig.

Grafik-Programmierung C128

Ein Bild gibt mehr Informationen als tausend Worte. Daß diese Aussage wahr ist, haben viele Computer-Anwender erkannt. Daher gehört die Programmierung von Grafik zu den interessantesten Aufgaben, die man mit dem Commodore 128 PC lösen kann.

Eine Hilfe für den Einsteiger und eine Fundgrube von Anregungen für den Profi soll dieses Buch sein. Das Themenfeld ist weit gespannt und behandelt unter anderem:

- Hochauflösende und Mehrfarbengrafik im C 128-Modus. Alle BASIC-7.0-Befehle dazu werden detailliert besprochen und deren Möglichkeiten und Grenzen gezeigt.
- Der Programmierung von Sprites und von Shapes ist ein weiterer Abschnitt gewidmet. Einige Verfahren zu ihrer Erstellung werden aufgezeigt.
- Des weiteren finden Sie nützliche Assemblerprogramme: beispielsweise eine OLD- und eine MERGE-Funktion, die die modulare Programmierung unterstützen.
- Die beiden Video-Chips (VIC und VDC) werden besprochen und ihre Programmierung im C 128-Modus vorgestellt. Natürlich ist auch die Grafik auf dem 80-Zeichen-Bildschirm ein Thema.
- Zum Erzeugen von selbstmodifizierenden Programmen lernen Sie die Technik kennen, mit der Sie leichter Funktionen untersuchen können und die es Ihnen ermöglicht, im BASIC-Programm Monitorbefehle zu verwenden.
- Ein interessantes Thema aus der neuesten Forschung sind die Fractals – merkwürdige Gebilde aus einer neuen Geometrie. Wie sie zustandekommen, was sie bedeuten und wie Sie diese auf dem 128 PC erzeugen können, zeigt Ihnen ein weiterer Abschnitt.

Viele Programme, Tabellen und Bilder erleichtern das Verstehen, denn: Ein Bild sagt mehr als tausend Worte.

ISB N 3-89090-202-2





DM 52,-
sFr. 47,80
öS 405,60



Heimo Ponnath

Grafik-Programmierung C128

Die faszinierende Welt der Grafik, erklärt an zahlreichen Anwendungsbeispielen in BASIC 7.0 und ASSEMBLER:

- Sprites
- Shapes
- Animation

Markt & Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Ponnath, Heimo:
Grafik-Programmierung C128 : d. faszinierende Welt der Grafik
erklärt an zahlr. Anwendungsbeispielen in BASIC u. Assembler:
Sprites, Shapes, Animation / Heimo Ponnath. –
Haar bei München : Markt-und-Technik-Verlag, 1986. –
ISBN 3-89090-202-2

15 14 13 12 11 10 9 8 7 6 5 4
89 88 87 86

ISBN 3-89090-202-2

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Vorwort

Als Besitzer eines Commodore 128 PC kommt sich manch einer, der alles ganz genau wissen möchte, vor als hätte er einen unerforschten Kontinent zu betreten. Hier und da gibt es bruchstückhaft Informationen, eigene Tests fügen ein Puzzleteil nach dem anderen hinzu, das Handbuch schließt auch noch einige Lücken. Insgesamt aber überwiegen die weißen Flecken auf der Karte dieses Kontinents Commodore 128 PC.

Grafik zu programmieren ist eine der interessantesten Aufgaben, die dem Benutzer gestellt werden können. Und neben dem Vergnügen an bunten Bildern findet sich hier auch eine wichtige Herausforderung, denn ein Bild sagt mehr als tausend Worte. Sobald aber ernsthaft Grafik per Computer betrieben wird, ist mehr Information über den Commodore 128 nötig als die bis jetzt vorhandene.

An dem vorliegenden Buch wurde buchstäblich bis zur letzten Minute gearbeitet, um Ihnen möglichst viele und aktuelle Fakten in die Hand zu geben. Sie werden das sicher daran merken, daß Ihnen hier kein Werk "aus einem Guß" vorliegt, sondern eine Zusammenstellung von Erkenntnissen, die durch intensive Arbeit am Commodore 128 PC nach und nach gewonnen wurden. Deshalb ist auch ein reichhaltiges Stichwortregister angefügt, das Ihnen den Zugang zum Inhalt erleichtern soll.

An welchen Leserkreis wende ich mich? Die Bandbreite ist sehr groß: Dem Einsteiger sollen die Erklärungen der Grafikbefehle in den ersten beiden Kapiteln helfen. Diese Erklärungen gehen nach und nach immer tiefer, bis sie auch für den eingefleischten "freak" interessant werden. Für eben diesen erfahrenen Programmierer sind die letzten Kapitel über die Video-Chips und die Speicherorganisation ein Handwerkszeug. Das Kapitel über die Fractals behandelt ein Thema, das alle Leser faszinieren dürfte, und auch die Möglichkeiten des Tastaturpuffers und das "Sammelsurium"-Kapitel

geben allen Benutzern ein nützliches Instrumentarium zur Hand. überdies finden Sie auf der beigefügten Diskette alle im Buch erläuterten Programme. Der Sinn dieser Arbeit sollte ja nicht darin liegen, daß Sie sich die Finger wund tippen bei der Eingabe der Beispiele.

Noch eins, bevor wir beginnen: Dieses Buch handelt nicht vom C64-Modus unseres Computers. Ich habe mich oft schon geärgert, wenn ich Literatur suchte zum Commodore 128 PC. Da war nämlich manches Mal die Hälfte des Buches dem C64-Modus gewidmet. Zum C64 aber gibt es genug Literatur. Sollten Sie also mehr darüber wissen wollen, dann können Sie anderweitig ins volle greifen. Es gibt kaum Unterschiede zwischen dem C64 und dem C64-Modus des Commodore 128 PC. Wo solche Unterschiede die Grafikprogrammierung betreffen, werden sie hier auch erläutert.

Die große Geduld und Nachsicht, die mir meine Frau und meine Tochter bewiesen haben, möchte ich hier dankend erwähnen. Dem Buchverlag Markt&Technik sei für die tatkräftige Hilfe, den Redaktionen des 64er-Magazin und von **happy computer** für ihre Kooperation gedankt.

1 Die Grafikbefehle

14 Befehle steuern im BASIC 7.0 des Commodore 128 die Grafikmöglichkeiten im engeren Sinn. Zählt man noch die Sprite- und Shapeanweisungen hinzu, dann verfügen wir gar über 27 grafische Werkzeuge in BASIC. Die Programmierung von Sprites und von Shapes wird an anderer Stelle behandelt. Hier geht es nun um die hochauflösende und die Mehrfarbengrafik, die wir uns im Detail ansehen werden.

1.1 GRAPHIC

Dieser Befehl dient - der Name sagt es schon - zum Umschalten in die verschiedenen Grafik- und Textmodi unseres Computers. Mittels

GRAPHIC A, B, C

können je nach den Kennungen A, B und C eine ganze Reihe von Optionen angewählt werden.

Die Kennung A wählt den Grafik-Modus aus:

A = 0	Der Textbildschirm - und zwar derjenige mit 40 Zeichen Zeilenbreite - wird eingeschaltet.
A = 1	Die hochauflösende Grafik wird aktiviert.
A = 2	Ein "Splitscreen", also ein Bildschirm, auf dem sowohl 40-Zeichen-Text als auch hochauflösende Grafik zulässig sind, kann damit eingeschaltet werden.
A = 3	Die Mehrfarbengrafik (auch Multicolorgrafik genannt) ist angewählt.
A = 4	Auch dieser Parameter erlaubt ebenfalls die Einrichtung eines Splitscreens. Diesmal erscheint zum 40-Zeichen-Text noch die Mehrfarbengrafik.
A = 5	schaltet den 80-Zeichen-Text ein.

Der Parameter B entscheidet darüber, ob der ausgewählte Bildschirm gelöscht wird oder nicht:

B = 0	Es wird nicht gelöscht.
B = 1	Der Bildschirm (in den Grafik-Modi 1 bis 4 auch die Bit-Map) wird gelöscht.

Der Parameter C spielt lediglich in den Modi 2 und 4 eine Rolle (Splitscreen). Er bestimmt, an welcher Zeile der Textbildschirm eingeblendet wird. Die splitscreens sind so eingerichtet, daß immer von dieser Zeile an abwärts Textbereich existiert. Voreingestellt - und daher muß C nicht immer angegeben werden - ist die Zeile 19.

Worin unterscheiden sich diese Grafik-Modi? Wenn Sie im C64-Modus Grafik programmieren wollen, dann kommen Sie ohne einen sehr tiefen Einblick in die Speicherorganisation des Computers nicht sehr weit. Auch wenn uns das der Commodore 128 durch sein leistungsfähigeres BASIC erspart; ein wenig sollten Sie dennoch wissen.

1.1.1 Die Grafik-Modi

1.1.1.1 40-Zeichen-Text

Vermutlich ist Ihnen bekannt, daß wir 2 Speicherbereiche zur Darstellung von Text auf dem 40-Zeichen-Bildschirm benutzen:

1024	bis	2023	Bildschirm-RAM und
55296	bis	56295	Bildschirm-Farben-RAM.

In jeweils 1000 (nämlich 40 Spalten mal 25 Zeilen) Speicherstellen liegt hier das fest, was uns der Monitor anzeigt. Die Aufteilung dieser beiden Speicher finden Sie in Ihrem Handbuch auf den Seiten 5-80 und 5-82. Wenn Sie dafür sorgen, daß in der linken oberen Ecke Ihres Bildschirmes ein A steht, dann finden Sie durch BANK0:PRINT PEEK(1024) die Zahl 1. Das ist der Bildschirmcode des Zeichens "A". Umgekehrt können Sie durch POKEs auch Zeichen in den Bildschirmspeicher schreiben:

```
BANK0:POKE1025,2
```

schreibt ein "B" an die zweite Position der ersten Zeile. Im Handbuch sind die Bildschirm-Codes der Zeichen zu finden. Der andere Speicherbereich ab 55296 versieht die Zeichen mit Farbe. In den alten Versionen des C64 mußten zwei POKEs geschrieben werden, um ein Zeichen sichtbar werden zu lassen. Der zweite POKE diente zum Einschreiben eines Farbcode in die korrespondierende Bildschirmfarbspeicherstelle. Hier ist das nicht nötig: Der Speicherbereich ab 55296 wird automatisch mit Farbcodes gefüllt: Die unteren 4 Bit enthalten nach dem Einschalten die Zahl 13, was dem Code der Farbe Hellgrün (vermindert um 1) entspricht. Es steht Ihnen aber frei, bunte Buchstaben zu erzeugen. Dazu ist lediglich ein anderer Farbcode in die zum Bildschirmspeicher gehörende Farbzelle zu schreiben. Das Bild 1.1 soll dieses Zusammenspiel beider Speicherbereiche verdeutlichen:

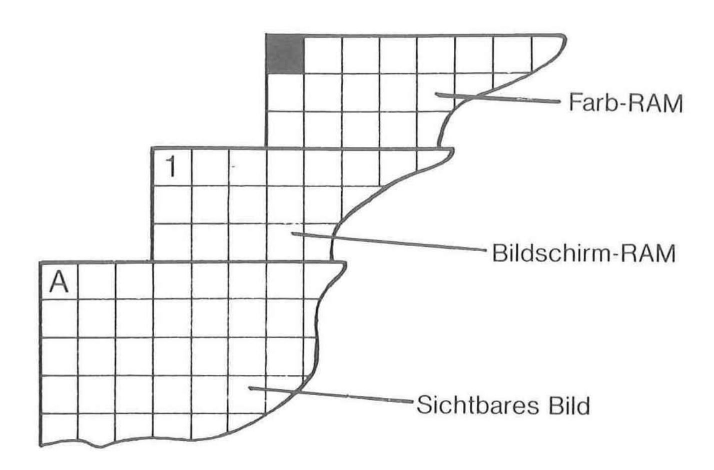


Bild 1.1: Ein Zeichen auf dem Bildschirm setzt sich aus der Bildschirm- und der Farbinformation zusammen

Übrigens, falls Ihnen die aktuelle Cursorfarbe nicht mehr gefallen sollte, dann können Sie sie durch

BANK0:POKE241,Farbcode

ändern. Auch alle danach geschriebenen Zeichen erhalten die so gewählte neue Farbe.

Nun wissen wir zwar, daß zum Auftreten eines Zeichens auf dem Bildschirm ein Zeichencode im Bildschirmspeicher und ein Farbcode im dazugehörigen Farbspeicher nötig sind, so ganz befriedigend ist das als Erklärung aber noch nicht. Woher weiß unser Computer eigentlich, was er abbilden soll, wenn da beispielsweise der Zeichencode 1 in Speicherstelle 1024 steht? Hier ist die Antwort: Für jedes Zeichen, das der Computer darstellen kann, gibt es im sogenannten Character-ROM ein Muster. Der Zeichencode dient als Zeiger auf das erste Byte eines solchen Musters. Die 1 deutet auf das erste Zeichenmuster, das zum großen Buchstaben "A" gehört. Es gibt übrigens auch das nullte Zeichen, den "Klammeraffen".

Jedem Zeichen sind im Character-ROM acht Byte zugeordnet. Ein Beispiel zeigt Bild 1.2:

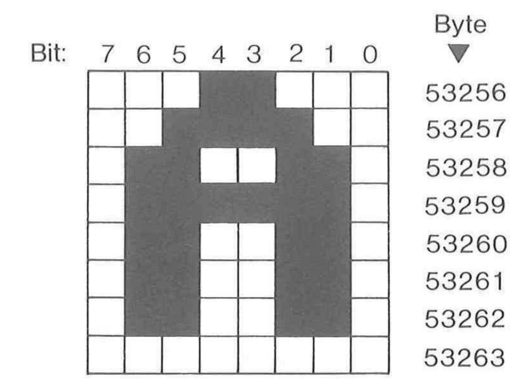


Bild 1.2: Das Zeichenmuster des Buchstaben A

Daraus ist zu ersehen, daß jedes Zeichen als gesetzte oder leere Punkte in einem 8mal8-Rasterfeld definiert ist. Falls Sie über einen guten Monitor verfügen und Kontrast sowie Helligkeit entsprechend einstellen, dann können Sie diese Punkte auch auf dem Bildschirm erkennen.

Zwei Fragen stellen sich aus dieser Erkenntnis heraus:

- Kann man eigene Zeichen erstellen und verwenden?
- Gibt es eine Möglichkeit, statt jeweils auf solch ein 8mal8-Raster auch auf einzelne Punkte einzuwirken?

Die erste Frage wird im Abschnitt über den VIC-Chip beantwortet werden. Hier sei nur erwähnt, daß das möglich ist.

Die zweite Frage führt uns zu den Grafik-Modi.

1.1.1.2 Der Hochauflösungs-Modus

Wir hatten festgestellt, daß wir im Textmodus (40-Zeichen) immer nur (beispielsweise durch POKE1024,1) auf ein 8mal8-Punkte-Raster zugreifen können. Die Auflösung unseres Bildschirmes beträgt in diesem Zustand 40 mal 25. Es gibt noch einige Tricks, eine sogenannte Viertelpunktgrafik zu entwerfen, in der die auf der Tastatur vorgegebenen Grafikzeichen (beispielsweise mit Commodore-Taste und C zu erhalten) geschickt verwendet werden. Es juckt aber in den Fingern, jeden Bildpunkt eines solchen

8mal8-Pixels einzeln anzusprechen. Das ist tatsächlich möglich mittels des Grafikmodus 1 (und auch 2), der durch GRAPHIC1 anwählbar ist. Damit erhöht sich die Auflösung ganz dramatisch:

320 horizontale und 200 vertikale Bildschirmpositionen stehen uns hier zur Verfügung. Wir haben den sogenannten Bit-Map-Modus eingeschaltet.

Das Prinzip ist dabei folgendes: Jedem Bildpunkt entspricht ein Bit eines Speicherbereiches, der Bit-Map. Ist in dieser Bit-Map ein Bit gleich 1, dann erscheint an der dazugehörigen Bildschirmposition ein Punkt (das ist wie das Eintragen von Geländeeinzelheiten in eine Landkarte, daher der Name "Map" = "Landkarte"). Jeweils 8 Bit ergeben ein Byte, woraus sich auf die Größe der Bit-Map schließen läßt: (320*200)/8 = 8000 Byte.

Alles, was auf dem Bildschirm erscheint, ist der Inhalt dieser Bit-Map, die durch den GRAPHIC1-Befehl eingerichtet wird. Wo befindet sie sich? Wenn Sie mal vor und nach dem Einrichten der Bit-Map durch PRINT FRE(0) nach dem freien Speicherplatz fragen, dann erkennen Sie es: In der Bank 0, also dem BASIC-Textspeicher, wird einiges verändert. Normalerweise (also ohne Bit-Map) beginnt der BASIC-Text in Speicherstelle $1C00 (das ist dezimal 7168). GRAPHIC1 schiebt den Textstart (und ein eventuell schon im Speicher befindliches Programm) in Windeseile nach oben bis $4000 (das ist dann dezimal 16384). Die Bit-Map beginnt ab $2000 (was dezimal 8192 entspricht).

GRAPHIC1 legt aber nicht nur eine Bit-Map an und schaltet den Videochip VIC auf den speziellen Grafik-Modus um, sondern definiert auch ein Koordinatensystem auf dem Bildschirm, das sich aus der Bit-Anordnung des Bildes ergibt. Sehen Sie dazu das Bild 1.3:

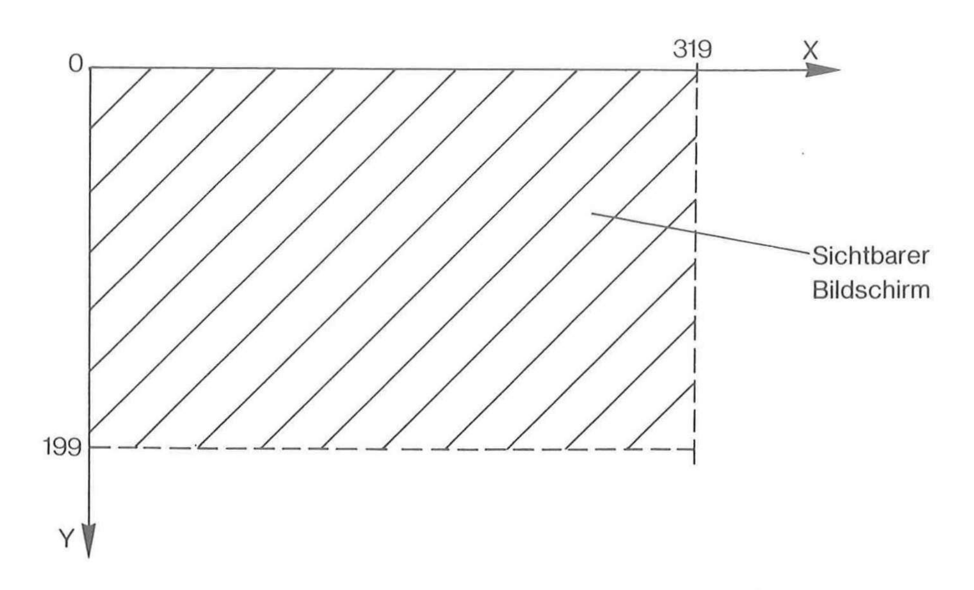


Bild 1.3: Die Koordinaten im Hochauflösungsmodus

Wir finden einen Koordinatenursprung in der linken oberen Bildecke, eine nach unten weisende Y-Achse und eine nach rechts verlaufende X-Achse: Man nennt solch ein System ein linkshändiges, was dem Anwender manchmal einige Probleme bereitet, weil das in der Mathematik übliche ein rechtshändiges ist (da wäre dann der Ursprung unten links zu finden, und die Y-Achse verliefe nach oben). Alle Zeichenbefehle beziehen sich im Normalfall auf die so festgelegten Koordinaten. Ein Befehl

```
DRAW1,0,0TO319,199
```

(zu den Einzelheiten kommen wir noch) zeichnet also eine Linie von links oben nach rechts unten.

Wir haben noch nicht erwähnt, wie der Computer die Farben in diesem Modus festlegt. Das soll beim COLOR-Befehl behandelt werden. Auf Koordinatensysteme (ja, in der Mehrzahl!) kommen wir beim SCALE-Befehl noch einmal zurück.

1.1.1.3 Die Mehrfarbengrafik

Während in den Grafik-Modi 1 und 2 jedes gesetzte Bit in der Bit-Map einen Bildpunkt verursacht, ist die Mehrfarbengrafik (Modi 3 und 4) immer auf Bit-Paare orientiert. Die Bit-Map hat zwar als Karte des Bildschirminhaltes ihre Bedeutung beibehalten, aber sie wird vom VIC-Chip nun anders gelesen. Es existieren 4 Möglichkeiten von Bitkombinationen je zweier Bit:

00	Hintergrund
01	Farbe 1
10	Farbe 2
11	Farbe 3

Auf welche Weise welche Farbe angesprochen wird, soll uns beim COLOR-Befehl weiter beschäftigen. Hier interessiert uns die Auswirkung auf die Auflösung.

Weil in der X-Richtung nun nur noch Bitpaare zählen, halbiert sich die Auflösung. Das Koordinatensystem der Multicolorgrafik ist ebenso aufgebaut wie das der hochauflösenden Grafik, lediglich in X-Richtung stehen uns (im Normalfall) nur noch 160 Positionen zur Verfügung. Bild 1.4 zeigt uns dieses neue Koordinatensystem:

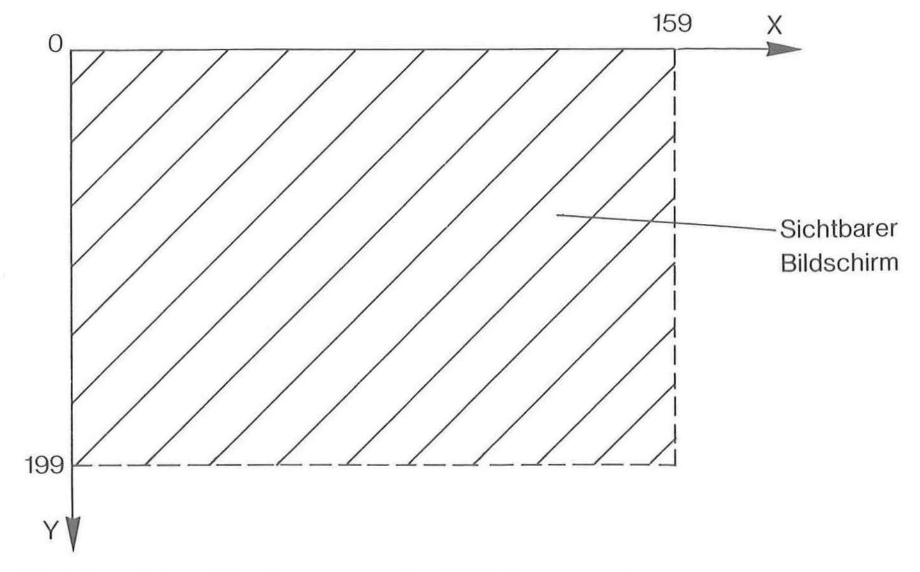


Bild 1.4: Das Koordinatensystem im Multicolormodus

1.1.1.4 Die Splitscreens

In den Grafik-Modi 2 und 4 wird der Bildschirminhalt durch jeweils zwei verschiedene Betriebsarten des VIC-Chips gewonnen. Im oberen Teil sehen wir den Inhalt der Bit-Map, im unteren aber 40-Zeichen-Text. Dies wird erreicht durch die sogenannte Rasterzeilenunterbrechung. Auch im C64-Modus kann mittels eines Maschinenprogramms diese Aufteilung des Bildschirmes programmiert werden. Dabei gab es immer ein Problem, das mit der Bearbeitungszeit zusammenhing. Die Rasterzeilenunterbrechung und die normale Unterbrechung (die 60 mal pro Sekunde stattfindet) durften sich nicht gegenseitig in die Quere kommen. Diese Schwierigkeit konnte in Programm-Betrieb meistens ganz gut gelöst werden. Im Direktmodus führten aber Cursorbewegungen und die Arbeit mit Sprites oft zu einer zittrigen Grenze der beiden Darstellungsmodi auf dem Bildschirm. Dasselbe Problem scheint im Commodore 128-Modus dafür verantwortlich zu sein, daß diese Grenze manchmal geradezu extrem gestört wird, beispielsweise beim Steuern von Sprites über die Splitscreen-Grenze im Grafikmodus 4.

Beim Einschalten eines Splitscreens (nochmal zum Wort: "screen" heißt auf deutsch "Bildschirm", und "split" kann übersetzt werden als "aufteilen"), beim Einschalten also befindet sich der Textcursor nicht automatisch im Textbereich. Er muß erst dorthin gefahren werden. Zwei Möglichkeiten bieten sich hier an: Zum einen kann ein PRINT AT simuliert werden durch:

```
BANK15:SYS65520,,Zeile,Spalte
```

Zum anderen bietet sich die Definition eines Bildschirmfensters an unter Benutzung des WINDOW-Befehls. Man legt dann einfach den gesamten Textbereich als WINDOW fest. Ist also keine spezielle Trennzeile angegeben worden (dann tritt der übergang von Grafik zu Text in Zeile 19 ein), dann kann das durch

```
WINDOW0,20,39,24
```

geschehen. Übrigens bleibt das Bildschirmfenster auch nach einer Rückkehr in den Textmodus erhalten. Wird das nicht gewünscht, kann durch

```
WINDOW0,0,39,24
```

der gesamte Bildschirm wieder benutzt werden.

1.1.1.5 Der 80-Zeichen-Textmodus

Sollten Sie stolzer Besitzer eines 80-Zeichen-Monitors sein, dann können Sie durch GRAPHIC5 die Text-Ausgaben auf dessen Bildschirm lenken. Interessanterweise ist es nun möglich, zwei getrennte Anzeigen zu erhalten, beispielsweise die Multicolorgrafik auf dem 40-Zeichen-Schirm und die Textausgaben auf dem 80-Zeichen-Monitor. Die Splitscreen-Optionen braucht man dann nicht mehr, sie sind sogar hinderlich, weil sie die Ausgabe von Text wieder auf den 40-Zeichen-Schirm lenken und uns die Mühe machen, den verringerten Grafikausgabebereich zu berücksichtigen (die Y-Achse wird ja nun ab Zeile 19 überdeckt vom Textbereich).

Die Textdarstellung auf dem 80-Zeichen-Schirm folgt etwa dem gleichen Prinzip, wie wir es schon beim 40-Zeichen-Text kennengelernt haben, mit folgenden Unterschieden:

Zunächst benutzt unser Computer jetzt einen anderen Videobaustein, den VDC 8563. Wir werden diesen Videochip an anderer Stelle noch etwas genauer kennenlernen.

Der Bildschirmspeicher ist nirgendwo im Bereich der 128K unseres Computers zu finden, die wir direkt ansprechen können (durch PEEK oder mittels Monitor). Ebensowenig werden wir einen Hinweis auf den Farbspeicher finden. Woran liegt das? Immerhin brauchen wir doch 80*25 Byte, also 2000 für das Bildschirm-RAM und ebenso viele Byte für die Farbzellen. Da sind wir auf eine Besonderheit des VDC-Chips gestoßen: Er benutzt einen eigenen 16K-RAM-Bereich, der völlig abgetrennt vom normalen Speicher existiert und von uns nur über 2 Speicherstellen ansprechbar ist. Welche das sind und wie man sie benutzt, soll ebenfalls beim VDC besprochen werden. Hier nur die Aufteilung dieses Speichers:

0000	bis	2047	Bildschirm-RAM
2048	bis	4095	Attribut-RAM
4096	bis	8191	frei
8192	bis	16385	Zeichenspeicher

Zur Bedeutung des Bildschirm-RAM braucht nichts mehr gesagt werden. Das Attribut-RAM entspricht etwa dem Bildschirmfarbspeicher, birgt aber noch mehr Möglichkeiten. Weil wir alle Aspekte der Farbgebung der anderen Modi beim COLOR-Befehl behandeln werden, mit diesem Befehl aber das VDC-Attribut-RAM nicht erschöpfend erfaßt werden kann, soll hier diese VDC-Spezialität erklärt werden. Zuvor noch ein paar Worte zum Zeichenspeicher. Während im 40-Zeichen-Textmodus ein Zeichenmuster aus

dem Character-ROM entnommen wird, ist das im 80-Zeichen-Textmodus nicht der Fall. Vielmehr findet hier schon in der Einschaltphase ein Kopieren aller Zeichen in den speziellen VDC-RAM statt. Von dort stammen nun auch die Zeichenmuster. Sie kennen sicher schon die Erscheinung, daß beispielsweise beim Umschalten auf den DIN-Zeichensatz (mit der CAPS-LOCK-Taste) einige Zeit verstreicht, bis alle Zeichen im neuen Look zu sehen sind: Das liegt daran, daß der gesamte Zeichensatz umkopiert werden muß. Im Bereich 8192 bis 16385 finden sich danach nur noch die DIN-Zeichenmuster.

Es wird sicher Ihrer Aufmerksamkeit nicht entgangen sein, daß hier die Rede von RAM ist im Gegensatz zum Zeichen-ROM, das bei der 40-Zeichentextdarstellung eine Rolle spielt. Wenn wir wissen, wie man dieses VDC-RAM ansprechen kann, wird uns nichts daran hindern, auch in den Speicherbereich des Zeichensatzes hineinzuschreiben. So können dann eigene Zeichen definiert werden.

Kommen wir nun aber zurück zum Attribut-RAM. Jeder Bildschirmspeicherstelle entspricht ein Byte dieses Attributbereiches. Durch dieses Byte aber wird noch mehr gesteuert als die Farbe, was Sie am Aufbau solch einer Speicherstelle erkennen können (siehe Bild 1.5):

Bit:	7	6	5	4	3	2	1	0
Name:	ALT	RVS	UL	FL	R	G	B	J

Bild 1.5: Aufbau eines Attribut-RAM-Bytes

Die Bit 0 bis 3 bestimmen die Farbe, was wir gleich noch untersuchen wollen.

Bit 4 : Ist hier eine 1 zu finden, dann blinkt das Zeichen.

Bit 5 : Das Zeichen ist unterstrichen, wenn hier eine 1 enthalten ist.

Bit 6 : Hiermit wird ein Aspekt der Zeichendarstellung doppelt gemoppelt: die Revers-Darstellung. Eigentlich kann nämlich durch Einschreiben einer 1 an diese Stelle genau dasselbe er-

reicht werden wie durch das ansonsten etwas speicherfressende Verfahren eines gesonderten Zeichenmusters, mit dem momentan die inversen Zeichen gezeichnet werden.

Bit 7 : Im Gegensatz zum 40-Zeichen-Textmodus, der die Groß- und Kleinschreibung nur durch Umschalten (per Commodore- und Shift-Taste) erlaubt, können im 80-Zeichenmodus beide Zeichensätze gleichzeitig verwendet werden. Welches von den beiden möglichen Zeichen auf dem Bildschirm zu sehen ist, entscheidet dieses Bit. Liegt hier eine 1 vor, dann stammt das sichtbare Muster aus dem 2. Zeichensatz.

Nun zu den Farb-Bits. Im Bild 1.5 sind diese mit R, G, B und I bezeichnet. Dabei steht R für Rot, G für Grün, B für Blau und schließlich I für Intensität. Je nach Mischung der Signale, die durch unterschiedliche Bit-Belegungen der 4 Bit erzeugt werden, ergeben sich die 15 möglichen Farbdarstellungen. Bild 1.6 zeigt alle Kombinationen:

Bit:	3	2	1	0			
	R	G	B	J	Dez.	Farbe	Entspricht Farbcode im 40-Z.-Modus
	0	0	0	0	0	schwarz	1
	0	0	0	1	1	d'grau	12
	0	0	1	0	2	blau	7
	0	0	1	1	3	h'blau	15
	0	1	0	0	4	grün	6
	0	1	0	1	5	h'grün	14
	0	1	1	0	6	türkis	4
	0	1	1	1	7	orange	9
	1	0	0	0	8	rot	3
	1	0	0	1	9	h'rot	11
	1	0	1	0	10	lila	13
	1	0	1	1	11	violett	5
	1	1	0	0	12	braun	10
	1	1	0	1	13	gelb	8
	1	1	1	0	14	h'grau	16
	1	1	1	1	15	weiß	2

Bild 1.6: Zuordnung der Bit-Kombination zu den Farben (stark vom verwendeten Monitor abhängig!)

All diese einzelnen Aspekte, die das Attribut-RAM erlaubt, können durch CHR$- oder/und ESC-Befehle angesteuert werden. So erzeugt

PRINT CHR$(2)"A"

ein unterstrichenes und

PRINT CHR$(27)+"F";"A"

ein blinkendes A. Das Handbuch gibt ab Seite 4-4 erschöpfend darüber Auskunft.

Eine kleine Besonderheit ist allerdings vergessen worden:

Falls Sie einmal die Control- und die S-Taste zusammen drücken, während ein Programm läuft, hält dieses sofort an. Damit haben wir eine Pausentaste im Computer. Erst nach einem beliebigen anderen Tastendruck arbeitet der Computer weiter.

1.1.2 GRAPHIC CLR

Eine Version des GRAPHIC-Befehls wurde Ihnen bisher noch verschwiegen. Mit

GRAPHIC CLR

führen Sie den Computer wieder in den reinen Textmodus zurück. Sie werden sagen, daß dazu doch die Befehle GRAPHIC0 oder GRAPHIC5 ausreichen. Das stimmt aber nur in Grenzen. Falls einmal einer der Grafik-Modi 1 bis 4 eingeschaltet wurde, damit also der ganze Speicher in Bank 0 verändert ist, bleibt die Bit-Map erhalten und der BASIC-Start bei $4000. Das erkennen Sie beispielsweise daran, daß - falls Sie nicht mit einer der Löschfunktionen das Bitmuster gelöscht haben - durch beispielsweise GRAPHIC1 auf dem Bildschirm immer noch das Grafikbild auftaucht, das Sie vorher einmal erstellt haben.

Erst durch den Befehl GRAPHIC CLR krempeln Sie die Bank 0 unseres Speichers wieder völlig um. Der BASIC-Start wandert wieder nach $1C00, die Bit-Map wird überschrieben.

1.2 COLOR

Wir kommen nun zur Farbgebung in den verschiedenen Modi. Im Befehl

COLOR A,B

bedeutet A das Folgende:

A = 0	Hintergrund im 40-Zeichen-Textmodus und in den Grafikmodi
A = 1	Vordergrundfarbe in den Grafikmodi. Im Multicolormodus wird damit die Farbe der Bit-Kombination 01 belegt.
A = 2	Multicolor-Farbe 1. Das betrifft die Farbe der Bit-Kombination 10.
A = 3	Multicolor-Farbe 2, die sich auf die Bit-Kombination 11 bezieht
A = 4	Farbe des Bildschirm-Rahmens
A = 5	Zeichenfarbe im Textmodus. Damit wird sowohl im 40- als auch im 80-Zeichen-Text die Zeichenfarbe bestimmt.
A = 6	Hintergrund des 80-Zeichen-Bildschirmes

B ist der Farbcode. Je nach Fabrikat, eingestelltem Kontrast, Helligkeit und anderen Gerätespezifica variiert die dargestellte Farbe. Der Code und die angegebenen Farben dienen daher eher als ungefährer Richtwert. Bild 1.7 zeigt die Zuordnung der verschiedenen Werte:

Farbe	Code	Farbe	Code
Schwarz	1	Hellbraun	9
Weiß	2	Braun	10
Rot	3	Rosa	11
Türkis	4	Dunkelgrau	12
Violett	5	Grau	13
Grün	6	Hellgrün	14
Blau	7	Hellblau	15
Gelb	8	Hellgrau	16

Bild 1.7: Farben und Codenummern

1.2.1 Die Farbspeicher

Sowohl einzelne Speicherstellen als auch größere RAM-Bereiche spielen eine Rolle bei der Farbzuordnung. Einen RAM-Bereich haben wir schon beim Textmodus erwähnt: den Bildschirmfarbenspeicher zwischen 55296 und 56295 ($D800 bis DBE7). Anders als beim VDC-Attribut-RAM spielen hier nur die Bits 0 bis 3 eine Rolle. Die oberen 4 Bit sind unbenutzt und unterliegen einem ständigen Wandel. Versuchen Sie mal, mittels Monitor und dem Kommando M FD800 mehrmals hintereinander in dieses Farb-RAM zu sehen. Im Einschaltzustand finden Sie allerlei Werte zwischen 0D und FD, denen aber allen das niedrigwertige Nibble D eigen ist.

Als weiterer RAM-Bereich liegt - bisher noch nicht erwähnt - in den Grafikmodi 1 bis 4 ein Farbspeicher für die Bit-Map vor, der von 7168 bis 8167 ($1C00 bis 1FE7) reicht. Parallel zur Einrichtung der Bit-Map sorgen die GRAPHIC-Befehle auch für die Zuordnung dieses Farb-RAM. Wie Sie sehen, enthält der Farbspeicher lediglich 1000 Byte Speicherplatz. Die Farbzuordnung geschieht also weiterhin wie im Text-Modus, nämlich Pixel für Pixel (wobei jedes Pixel in der Form des 8mal8-Rasters gehandhabt wird). So können Sie zwar auch im Hochauflösungsmodus in verschiedenen Farben auf den Bildschirm zeichnen (einfach, indem COLOR1 vor dem Zeichenbefehl eine andere Farbe erhält), sobald aber die Zeichnung genügend dicht an eine andere heranreicht, wechselt an der Stelle das gesamte Pixel seine Farbe, also auch der darin verlaufende Teil der alten Zeichnung. Das ist gut zu erkennen im Verlauf des Programmes "2D DARSTELLUNGEN", das Sie auf der beiliegenden Diskette finden (es wird an anderer Stelle - bei selbstmodifizierenden Programmen - besprochen).

Der Aufbau der einzelnen Byte dieses RAM-Bereiches hängt von der Art des eingeschalteten Grafik-Modus ab. In der Hochauflösungsgrafik enthält das LSN (das sind die Bits 0 bis 3, also das untere Nibble) einen Code für die Hintergrundfarbe, wohingegen das MSN (die Bits 4 bis 7, das obere Nibble) den Code für die Vordergrundfarbe darstellt. Der verwendete Code ergibt sich jeweils aus dem Farbcode (vorhin als B bezeichnet) minus 1.

Im Mehrfarbenmodus dagegen finden wir im LSN die Multicolorfarbe 1 (ebenfalls als Farbcode - 1). Das MSN bleibt unverändert.

Übrigens ist dieser Inhalt des Farb-RAMs erst dann darin zu finden, wenn auf irgendeine Weise ein Löschvorgang stattgefunden hat, beispielsweise durch GRAPHIC1,1. Vorher - also dann, wenn dieser Speicher lediglich

durch GRAPHIC1 eingerichtet wurde, aber nicht gelöscht -, können Sie dort nur wirres Bytedurcheinander erkennen, falls Sie mittels Monitor-Kommando M dort hineinsehen.

Zwei Speicherstellen in der page 3 korrespondieren mit dem eben vorgestellten Farb-RAM:

$3E2 (dez.994)FG-BG

$3E3 (dez.995)FG-MC1

FG-BG hängt mit der hochauflösenden, FG-MC1 mit der Multicolor-Grafik zusammen. Im Einschaltzustand finden wir in diesen beiden Speicherzellen:

$3E2: $DB = bin.1101 1011

13 / 11
h'grün-1 /d'grau-1
Vordergr./Hintergr.

$3E3: $D1 = bin.1101 0001

13 / 1
h'grün-1 / weiß-1
Vordergr./Multicolor1

Mit diesen Werten wird das Farb-RAM belegt, sobald ein Grafik-Modus 1 bis 4 eingeschaltet wird und ein Löschvorgang stattfindet. Sobald aber durch einen COLOR-Befehl andere Farben gewählt werden, drückt sich diese Änderung in diesen beiden Speicherstellen aus, und der nächste Löschvorgang belegt das Farbram mit dem Inhalt der entsprechenden Speicherstelle.

Da ergibt sich allerdings ein Problem: Nehmen wir an, wir hätten den Grafik-Modus 1 eingeschaltet und wollen nun in den Multicolormodus(3) umschalten. Das ist zwar ohne weiteres möglich, aber die Farben werden erst dann umgeschaltet (also das Farbram erst dann mit dem Code aus $3E3 belegt), wenn ein Löschvorgang stattgefunden hat. Dann ist aber auch die Bit-Map gelöscht! Diese Schwierigkeit kann auf einigen Umwegen gelöst werden. Wie, das werden wir nachher noch sehen.

Vier Zeropage-Speicherstellen spielen in der Farbgebung eine gewichtige Rolle: $83 - $86 (dezimal 131 bis 134):

$83	COLSEL	aktuelle Farbe
$84	MULTICOLOR1	Multicolorfarbe 1
$85	MULTICOLOR2	Multicolorfarbe 2
$86	FOREGROUND	Vordergrundfarbe

Im Einschaltzustand findet man in $83 den Wert 0, in $84 eine 1 (also Farbcode von Weiß minus 1), in $85 steht 2 (das ist Rot -1), und in $86 liegt D (also 13, was Hellgrün -1 bedeutet).

COLSEL enthält nach jedem Grafikbefehl die darin angegebene Farbquellenziffer (die werden wir noch kennenlernen).

Aus dem C64-Modus sind Ihnen zwei Speicherstellen des VIC-Chip sicher bekannt:

$D020	dez.53280	Rahmenfarbe
$D021	dez.53281	Hintergrundfarbe

Auch im Commodore 128-Modus haben diese Speicherstellen diese Bedeutung. Im Einschaltzustand finden wir die Farben im LSN:

$D020	enthält $FD	= bin.1111 1101 h'grün-1
$D021	enthält $FB	= bin.1111 1011 d'grau-1

Nun kennen wir die beteiligten Speicher. Welche Auswirkungen hat der COLOR-Befehl?

1.2.2 Wie wirkt COLOR ?

Fassen wir die bis jetzt gewonnenen Erkenntnisse zur Farbgebung der Grafikbildschirme zusammen:

COLOR0,X	belegt den Hintergrund mit der Farbe X. Die Bits 0 bis 3 von $3E2 enthalten den Wert X-1. Die Bits 0 bis 3 von $D021 werden ebenfalls mit X-1 beschrieben.
COLOR1,X	bestimmt die grafische Vordergrundfarbe. In $86 erscheint X-1 Die Bits 4 bis 7 der Speicherstellen $3E2 und $3E3 enthalten danach X-1.
COLOR2,X	legt die Multicolorfarbe 1 fest.$84 hat nun X-1 zum Inhalt. In den Bits 0 bis 3 der Speicherstelle $3E3 befindet sich ebenfalls X-1.
COLOR3,X	bestimmt die Multicolorfarbe 2. In $85 befindet sich danach der Wert X-1.

Schließlich wird bei eingeschaltetem Grafik-Modus 1 bis 4 durch einen Löschvorgang der Speicherbereich $1C00 bis 1FE7 (dezimal 7168 bis 8167) beschrieben mit

- dem Inhalt von $3E2 bei den Modi 1,2

- oder dem Inhalt von $3E3 in den Multicolormodi 3 und 4.

Wir werden nun ausprobieren, welche Bitpaarkombination welcher Farbe im Multicolormodus entspricht. Dazu schreiben wir an den Anfang der Bit-Map drei Zahlen, die diese drei möglichen Kombinationen repräsentieren:

bin.1111 1111	=	dez.255 (Kombination 11)
bin.1010 1010	=	dez.170 (Kombination 10)
bin.0101 0101	=	dez. 85 (Kombination 01)

Die Kombination 00 entspricht der Hintergrundfarbe. Das Programm MULTICOLORTEST1 fragt zunächst nach den von Ihnen gewünschten Farben.

Programm MULTICOLORTEST1

Welche Bit-Kombination gehört zu welcher Farbquelle?

```
5 REM ***** MULTICOLORTEST 1 *****
10 INPUT"FARBEN F0,F1,F2,F3";F0,F1,F2,F3
20 COLOR0,F0:COLOR1,F1:COLOR2,F2:COLOR3,F3
30 GRAPHIC3,1:SCNCLR3
40 BANK0:POKEDEC("2000"),255:POKEDEC("2001"),255
50 POKEDEC("2002"),170:POKEDEC("2003"),170
60 POKEDEC("2004"),85:POKEDEC("2005"),85
70 CHAR1,10,10,"OBEN :11"
80 CHAR2,10,13,"MITTE:10"
90 CHAR3,10,16,"UNTEN:01"

READY.
```

Die Reihenfolge der Angaben folgt dabei den verschiedenen A-Kennungen des COLOR-Befehls, die wir vorhin verwendet haben. In Zeile 20 werden die von Ihnen gewünschten Farben dann in die Register geschrieben, in Zeile 30 der Mehrfarbenmodus eingeschaltet und das Farbram beschrieben. Die Zeilen 40 bis 60 tragen die oben ermittelten Bitkombinationen in die ersten Plätze der Bit-Map ein. Die folgenden CHAR-Befehle (die lernen wir noch kennen) schreiben nun jeweils in einer der drei Farben einen Orientierungstext auf den Bildschirm. Übrigens sieht der Text manchmal in diesem Modus etwas merkwürdig aus: die Zeichenmuster sind nämlich nicht für die Bitpaar-Darstellung entworfen.

Aus dem Programm folgt eine recht einprägsame Regel: Der Dezimalwert einer Bitkombination ist die Kennung A des dazugehörigen COLOR-Befehls. Erklärung:

Die Bitkombination 00 wird mittels COLOR0,... farblich festgelegt, die Bitkombination 10 (das ist dezimal 2) folgt in der Zeichnung der Farbe, die durch COLOR2,... definiert wird, und so weiter.

Nun können wir zusammenfassen:

Im hochauflösenden Modus folgt die Farbgebung dem Schema in Bild 1.8:

Bit:	7	6	5	4	3	2	1	0
bestimmt:	Vordergrund				Hintergrund			
Bit-Map-Inh.:	1				0			
Farbbefehl:	Color 1,...				Color 0,...			

Bild 1.8: Ein Byte aus dem Farb-RAM ab $1C00 in den Grafik-Modi 1 und 2

Im Mehrfarbenmodus kommt die Hintergrundfarbe (bei 00) aus der Speicherstelle 53281, die Farben der Kombinationen 01 und 10 werden gemäß dem Schema in Bild 1.9 verteilt:

Bit:	7	6	5	4	3	2	1	0
bestimmt:	Vordergrund				Multicolorfarbe 1			
Bit-Kombination:	01				10			
Farbbefehl:	Color 1,...				Color 2,...			

Bild 1.9: Ein Byte aus dem Farb-RAM ab $1C00 in den Grafik-Modi 3 und 4

Woher kommt die Farbe der Bit-Kombination 11? Wir haben festgestellt, daß diese Farbe in der Speicherstelle $85 festgehalten wird. Im C64-Modus wird sie aus dem normalen Text-Farb-RAM (ab $D800) entnommen. Überprüft man aber im Commodore-128-Modus diesen Speicherbereich, dann ist darin nur die normale Textfarbe zu finden. Hier muß also ein anderes Prinzip der Farbgebung verfolgt werden. Versuche zeigen, daß auch die Multicolorfarbe 2 pixelweise vergeben wird, was auf ein weiteres Farb-

RAM schließen lassen könnte, aber die Suche danach verläuft ergebnislos. Diese Frage kann erst dann geklärt werden, wenn man in das Bit 0 der Speicherstelle 1 eine 0 schreibt. Plötzlich erscheint im Monitorausdruck ab $D800 ein weiteres Farb-RAM: das der Multicolorfarbe 2.

1.2.3 Farbwechsel

Zwei Aspekte sind es nun noch, die in bezug auf Farbgebung interessieren:

- Kann man mehr als 4 Farben verwenden?
- Kann man in einem fertigen Bild die Farben ändern?

Der erste Aspekt ist sicher erfüllbar, denn ebenso, wie das Farb-RAM des Text-Bildschirmes durch POKE-Befehle Zelle für Zelle ansprechbar ist, ist es auch das Grafik-Farb-RAM ab $1C00. Allerdings muß man sich gut überlegen, was man hier hineinschreibt: Sowohl die Bits 0 bis 3 als auch die Bits 4 bis 7 haben ja eine Bedeutung, wie wir gesehen haben.

Eine andere Möglichkeit ist es, mittels des COLOR-Befehls eine oder mehrere Farben (Vordergrund, Multicolor1 oder 2) zu ändern und dann mittels eines Grafik-Befehls eben diese Farbquelle aufzurufen (dazu kommen wir noch). Hier ist zu beachten, daß sich die grafischen Objekte, die damit gezeichnet werden, nicht zu nahe kommen, weil ja immer ein ganzes Pixel umgefärbt wird und daher auch schon vorhandene Bildteile unter Umständen die neue Farbe annehmen.

Wesentlich schwieriger ist es allerdings, ein Bild, dessen Farbgebung uns nicht gefällt, umzufärben. Nehmen wir an, ein Multicolorbildschirm trüge Farben, die wir verändern wollen. Was wäre zu tun?

Die Farbe aller Bitpaare 00 läßt sich einfach durch COLOR0,... umschalten. Andere COLOR-Befehle für die 3 restlichen Farben oder auch für die Vordergrund- und die Hintergrundfarbe im hochauflösenden Modus fruchten nichts: Erst ein Löschvorgang ergäbe die Neubelegung des Farb-RAMs, wodurch aber auch die Bit-Map frei und unser Bild zerstört würde.

Da scheint nur eine Schleife Abhilfe zu bringen, die den Farbspeicher neu schreibt. Nehmen wir an, unsere neue Vordergrundfarbe hieße F1, und die neue Hintergrund- (in den Grafikmodi 1 und 2) oder Multicolorfarbe 1 (im Grafikmodus 3 oder 4) wäre F2. Dann haben wir zunächst die Gestalt eines Byte aus dem Farb-RAM zu bestimmen:

F1=F1-1:F2=F2-1

Das Byte wäre dann

16*F1+F2

Die Schleife lautete nun:

FOR I=7168 TO 8167:POKEI,16*F1+F2:NEXT I

So etwas dauert natürlich eine Weile, und wir können ganz gemächlich zusehen, wie von oben nach unten die Farbwechsel stattfinden.

Etwas schneller geht das mit einem Trick, der im Abschnitt zu den selbstmodifizierenden Programmen erklärt wird. Die beiden folgenden Programme MULTICOLORTEST2 und HIRES-FARBAEND zeigen Ihnen das Verfahren:

Programm MULTICOLORTEST2

Ein fertiges Bild ändert seine Farben

```
5 REM ***** MULTICOLORTEST 2 *****
10 INPUT"FARBEN F0,F1,F2,F3";F0,F1,F2,F3
20 COLOR0,F0:COLOR1,F1:COLOR2,F2:COLOR3,F3
30 GRAPHIC3,1:SCNCLR3
40 BANK0:POKEDEC("2000"),255:POKEDEC("2001"),255
50 POKEDEC("2002"),170:POKEDEC("2003"),170
60 POKEDEC("2004"),85:POKEDEC("2005"),85
70 CHAR1,10,10,"OBEN :11"
80 CHAR2,10,13,"MITTE:10"
90 CHAR3,10,16,"UNTEN:01"
100 PRINT"NEUE FARBEN FUER 1 UND 2:"
110 INPUT"F1,F2=";F1,F2:F1=F1-1:F2=F2-1:F=16*F1+F2
120 PRINTCHR$(147)CHR$(17)
130 PRINT"MONITOR"CHR$(17)CHR$(17)CHR$(17)CHR$(17)
140 PRINT"F 01C00 01FE7 "HEX$(F)
150 PRINT"X"CHR$(17)
160 PRINT"RUN200"
170 PRINTCHR$(19);
180 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE845,13:POKE208,4:END
190 :
200 PRINTCHR$(147)CHR$(17)"DAS WARS!"
```

Programm HIRES-FARBAEND

Ein Bild in den Grafik-Modi 1 und 2 ändert seine Farben

```
10 REM ***** HIRES-FARBAENDERUNGEN *****
20 INPUT"FARBEN F0,F1";F0,F1
30 COLOR0,F0:COLOR1,F1
40 GRAPHIC1,1:SCNCLR1
50 BOX1,10,10,100,100,0,1:BOX0,20,20,30,30,0,1
60 PRINT"NEUE FARBEN FUER 0 UND 1:"
70 INPUT"F0,F1=";F0,F1:F0=F0-1:F1=F1-1:F=16*F1+F0
80 PRINTCHR$(147)CHR$(17)
90 PRINT"MONITOR"CHR$(17)CHR$(17)CHR$(17)CHR$(17)
100 PRINT"F 01C00 01FE7 "HEX$(F)
110 PRINT"X"CHR$(17)
120 PRINT"RUN160"
130 PRINTCHR$(19);
140 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE845,13:POKE208,4:END
150 :
160 PRINTCHR$(147)CHR$(17)"DAS WARS!"
```

Beide Programme zeichnen zunächst etwas auf den Bildschirm und fragen dann nach den gewünschten neuen Farben. Falls Sie nur mit einem Bildschirm arbeiten, sollten Sie noch die Umschaltung auf den Textbildschirm (GRAPHIC0 vor der Frage) und hinterher wieder auf den Grafikbildschirm vornehmen. Nach dem Berechnen des Bytewertes schaltet das Programm den Monitor ein und füllt mittels der F-Funktion desselben den Farbspeicher von $1C00 bis 1FE7. Danach verläßt es den Monitor wieder und fährt mit dem BASIC-Programm fort (ab Zeile 160 bzw. 200).

Auf diese Weise können alle beiden Farben im Hochauflösungsmodus verändert werden. Drei Farben von Multicolorbildern sind ebenfalls auf diese Art und mit COLOR0,... zu erneuern. Interessant ist es übrigens, daß trotz des veränderten Farb-RAMs ein Zeichenbefehl (beispielsweise DRAW1,...) immer noch mit der alten Zeichenfarbe arbeitet. Die Erklärung liegt vermutlich darin, daß dazu die Speicherstellen ab $83 Verwendung finden. Wenn man auch die darin befindlichen Angaben der neuen Farbgebung anpaßt, geschieht das nicht mehr.

Wie schon vorhin angedeutet, ist die Änderung der Multicolorfarbe 2, von der noch nicht die Herkunft geklärt werden konnte, ein Problem. Bisher kenne ich da nur eine etwas umständliche Lösung: Speichern Sie Ihre Grafik inclusive des Farb-RAM ab. Das kann mittels des BSAVE-Befehls geschehen:

BSAVE"Bild",ON B0,P7168 TO P16191

Laden Sie nun das Bild wieder ein mittels:

COLOR3,X:GRAPHIC3,1:BLOAD"Bild"

In X steht dabei die neue gewünschte Multicolorfarbe 2, die nun gleich beim Einladen berücksichtigt wird.

1.3 DRAW

Nach all diesen etwas ermüdenden, aber notwendigen Vorbereitungen kommen wir nun zu den Zeichenbefehlen, deren einfachster der DRAW-Befehl ist. "To draw" ist englisch und kann übersetzt werden mit "zeichnen".

DRAW A,X1,Y1 TO X2,Y2

zeichnet eine Linie vom Punkt X1,Y1 zum Punkt X2,Y2 (Siehe Bild 1.10).

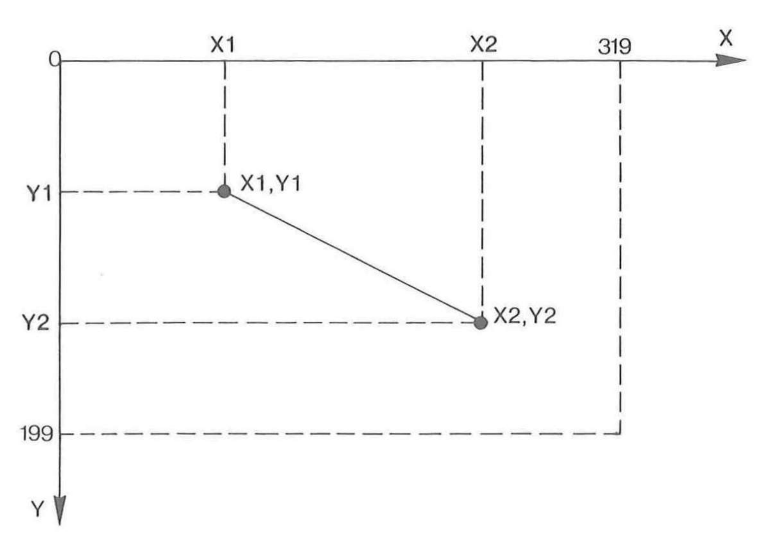


Bild 1.10: Die Wirkung des DRAW-Befehls

A ist eine Kennzahl für die Quelle, aus der die Farbe der Zeichnung herrührt. A hat dabei folgende Bedeutung:

A = 0	Hintergrundfarbe
A = 1	Vordergrundfarbe
A = 2	Multicolorfarbe 1
A = 3	Multicolorfarbe 2

Dieser Befehl ist sehr vielseitig verwendbar. Es ist beispielsweise möglich, einen Linienzug über weitaus mehr als nur 2 Punkte zu führen. Der Befehl

DRAW1,X1,Y1 TO X2,Y2 TO X3,Y3 TO X4,Y4 TO X5,Y5 TO X1,Y1

kann so ein grafisches Gebilde wie in Bild 1.11 erzeugen:

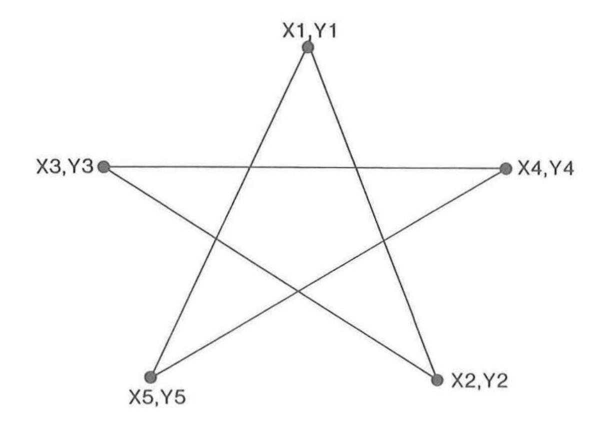


Bild 1.11 : Beispiel einer durch einen DRAW-Befehl erzeugten Figur

Lautet der Befehl

DRAW1,X1,Y1

dann wird lediglich der Punkt X1,Y1 gezeichnet. Schreibt man dagegen den Befehl als

DRAW1 TO X1,Y1

dann dient als Ausgangspunkt die aktuelle Grafik-Cursorposition. Den Grafik-Cursor werden wir beim LOCATE-Befehl noch kennenlernen. Von dieser Position aus wird dann eine Linie nach X1,Y1 gezogen.

Nach der Ausführung der verschiedenen DRAW-Befehle befindet sich der Grafik-Cursor an dem Punkt, der durch die letzte Koordinatenangabe charakterisiert wird.

Läßt man übrigens die Farbkennung weg, schreibt also beispielsweise

DRAW,X1,Y1 TO X2,Y2

dann wird die Linie in der Farbe gezeichnet, die im vorangegangenen Zeichenbefehl verwendet wurde. Vielleicht erinnern Sie sich noch an die Speicherstelle $83 COLSEL?. Dort befindet sich immer noch vom vergangenen Befehl her die Farbkennung.

1.4 BOX

Mit diesem Zeichenbefehl können Rechtecke erzeugt werden. Auch hier sind wieder eine ganze Menge Variationen möglich. Die Syntax des BOX-Befehls ist:

BOX A, X1,Y1,X2,Y2, W, F

A ist wieder - wie schon beim DRAW-Befehl - die Farbquelle. Ebenso wie dort kann man A auch weglassen, und das Rechteck erhält dieselbe Farbe wie das zuvor gezeichnete Grafik-Objekt.

Bild 1.12 erläutert die Bedeutung der Eck-Koordinaten X1,Y1 und X2,Y2:

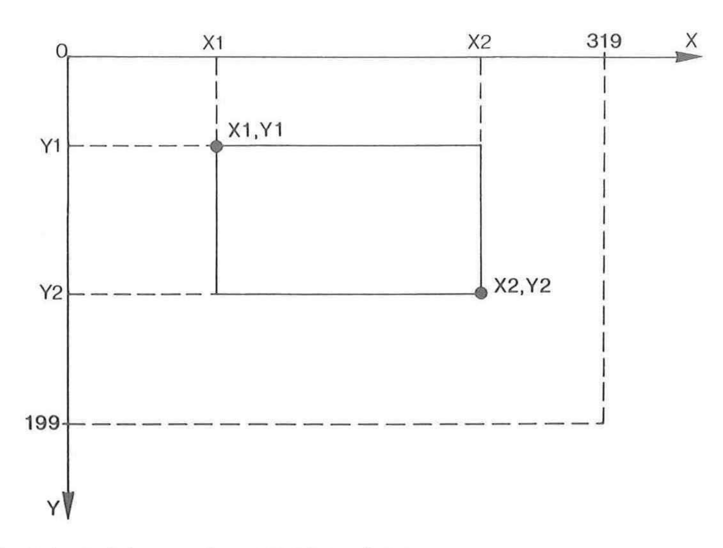


Bild 1.12: Die Wirkung des BOX-Befehls

X1,Y1 sind also die Koordinaten der linken oberen Ecke, X2,Y2 diejenigen der rechten unteren. Wir werden darauf gleich noch einmal zurückkommen.

W ist ein Winkel (in Grad angegeben!), um den das Rechteck zu drehen ist. Die Drehung erfolgt um den Mittelpunkt des Rechtecks im Uhrzeigersinn (siehe Bild 1.13).

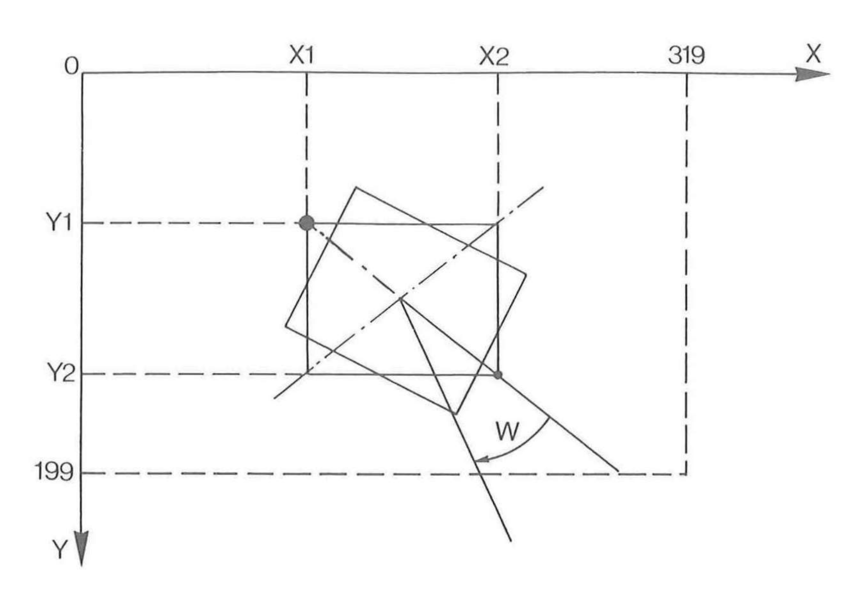


Bild 1.13: Drehung des Rechtecks durch den Parameter W im BOX-Befehl

W darf alle Werte zwischen 0 und 65535 annehmen.

F ist eine Kennung, die festlegt, ob das Rechteck ausgefüllt werden soll:

F = 0	Nicht ausfüllen
F = 1	In der gewählten Farbe ausfüllen

Interessant ist, was geschieht, wenn man Koordinaten eingibt, die unlogisch erscheinen. Beispielsweise in

```
BOX1,200,100,60,60
```

wurden die Eckpunkte vertauscht. Der Computer wählt sich selbst die richtigen Ecken aus und zeichnet die Gestalt, die wir bei der richtigen Reihenfolge erhalten hätten, also BOX1,60,60,200,100 Würfeln wir die Koordinaten vollends durcheinander (siehe Bild 1.14):

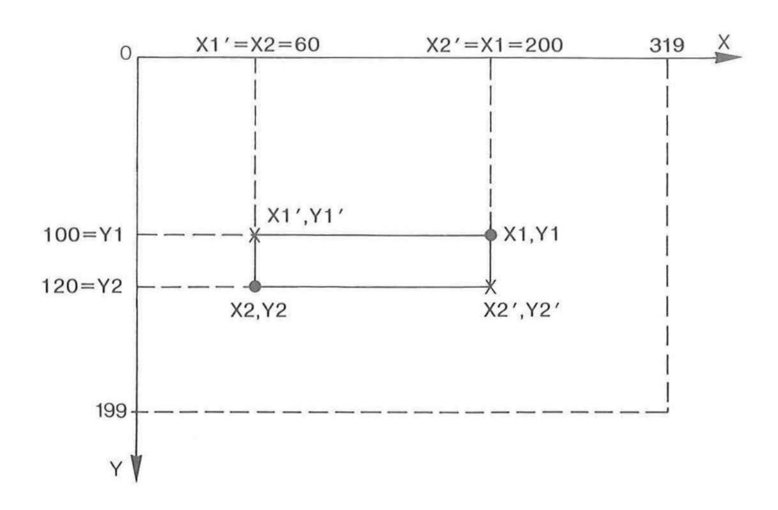


Bild 1.14: Durcheinandergewürfelte Koordinaten führen dennoch zum richtigen Ergebnis: BOX1,X2,Y2,X1,Y1 wird gezeichnet wie BOX1,X1',Y1',X2',Y2'.

BOX1,200,100,60,120 gibt die linke untere und die rechte obere Ecke an. Gezeichnet wird vom Computer dennoch das richtige Rechteck (mit den Ecken X1',Y1' und X2',Y2'), als hätten wir ihm befohlen BOX1,60,100, 200,120.

Es scheint, als gäbe es keinen Unterschied bei den verwürfelten Koordinatenangaben. Da existiert aber doch einer! Wir haben bisher den Grafik-Cursor nicht berücksichtigt. Der steht nämlich immer nach der Zeichnung am Punkt X2,Y2. Das bedeutet, daß er im Fall BOX1,200,100,60,120 auf dem Punkt 60,120 steht, wohingegen er im anderen Fall bei 200,120 zu finden wäre.

Im Fall einer Drehung des Rechteckes durch einen Wert W wird der Grafik-Cursor nicht mitgedreht. Er bleibt stur bei X2,Y2 stehen.

Noch nicht ausdrücklich erwähnt wurde die Möglichkeit, im Fall, daß sie nicht benötigt werden, die Größen W und F einfach wegzulassen. Sie werden dann beide als 0 angenommen. Man kann aber auch X2 und Y2 weglassen. Das zu zeichnende Rechteck setzt dafür einfach wieder die Koordinaten des Grafik-Cursors ein.

Ein Beispiel:

BOX3,100,50,200,80,30,1

zeichnet in der Multicolorfarbe 2 (dazu muß natürlich der Grafikmodus 3 eingeschaltet sein) ein Rechteck, das um 30 Grad um seinen Mittelpunkt im Uhrzeigersinn gedreht wurde und ausgefüllt ist.

1.5 CIRCLE

Ein geradezu phantastisch vielseitiger Befehl, nicht nur zum Zeichnen von Kreisen, wie es der Name (circle ist das englische Wort für Kreis) suggeriert. Was man damit alles anstellen kann, werden Sie gleich noch sehen. Wieder kann man über eine lange Parameter-Liste alle Optionen ausschöpfen. Aber keine Angst, es müssen bei weitem nicht immer alle angegeben werden. Komplett sieht der Befehl so aus:

CIRCLE A, XM,YM, RX,RY, W1,W2, W, S

Der erste Parameter A dient wie schon bei den anderen Zeichenbefehlen der Festlegung der Farbquelle.

Die Koordinaten XM,YM legen den Mittelpunkt des Objektes fest, das wir zu zeichnen gedenken. Soll es ein Kreis werden, ist es also der Kreismittelpunkt.

RX und RY sind die Halbachsen in horizontaler und vertikaler Richtung. Gedanklich geht man dabei von der Ellipse aus, die ja zum Kreis wird, wenn beide Halbachsen gleich sind. In diesem Fall ist RX = RY, und man kann sogar RY weglassen. Bild 1.15 zeigt, wie die bisher verwendeten Bezeichnungen zu verstehen sind:

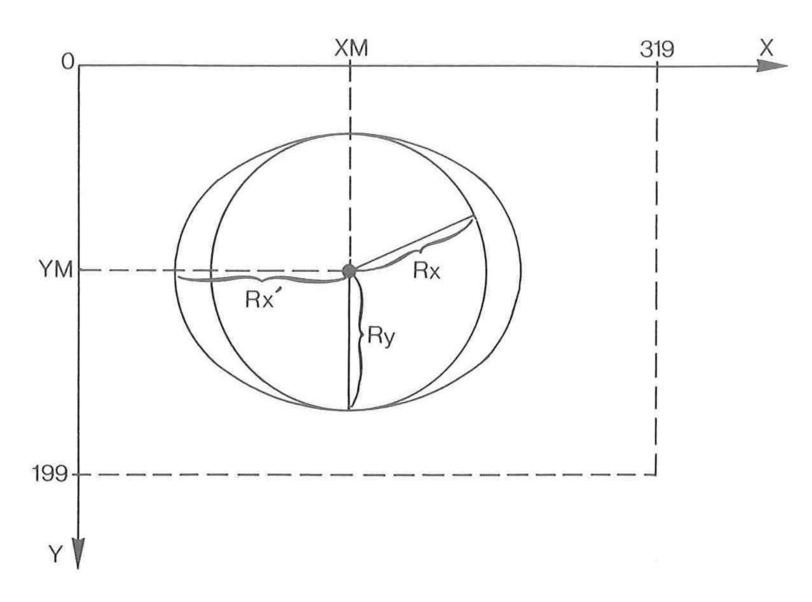


Bild 1.15: Wirkung der Befehle CIRCLE1,XM,YM,RX und CIRCLE1,XM,YM,RX',RY

Zwei CIRCLE-Befehle erzeugen hier einen Kreis (CIRCLE1,XM,YM,RX) und eine Ellipse (CIRCLE1,XM,YM,RX',RY).

Die Angaben W1 und W2 erlauben die Festlegung eines Bogens aus dem Gesamtobjekt, der zu zeichnen ist. Ausgangspunkt für die Messung der Winkel W1 und W2 ist der obere Scheitelpunkt der Ellipse, die man sich durch die Figur gelegt denken kann. Die Formulierung hört sich wesentlich einfacher an, wenn man von der Zeichnung eines Kreises ausgeht, der im normalen Koordinatensystem liegt. Dann zählt der Winkel vom obersten Kreispunkt an mit dem Uhrzeigersinn, was Bild 1.16 erläutern soll:

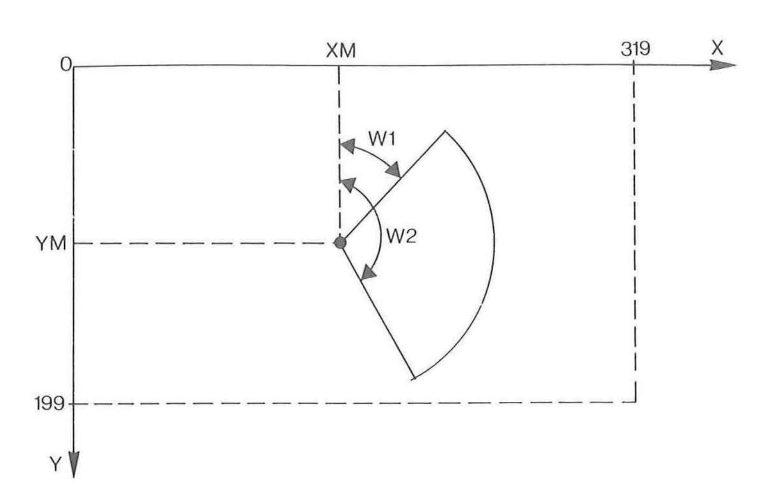


Bild 1.16: Zeichnen eines Kreis- (oder Ellipsen-)bogens durch CIRCLE1,XM,YM,RX,RY,W1,W2

W1 und W2 dürfen ohne Fehlermeldung alle Werte zwischen 0 und 65535 annehmen, wobei allerdings Angaben, die größer als 33024 sind, Unsinn auf dem Bildschirm erzeugen, falls sie nicht ohnehin schon unsinnig sind.

Was geschieht, wenn W1 größer als W2 gewählt wird? Kein Problem: Das Ergebnis finden Sie in einem Beispiel in Bild 1.17:

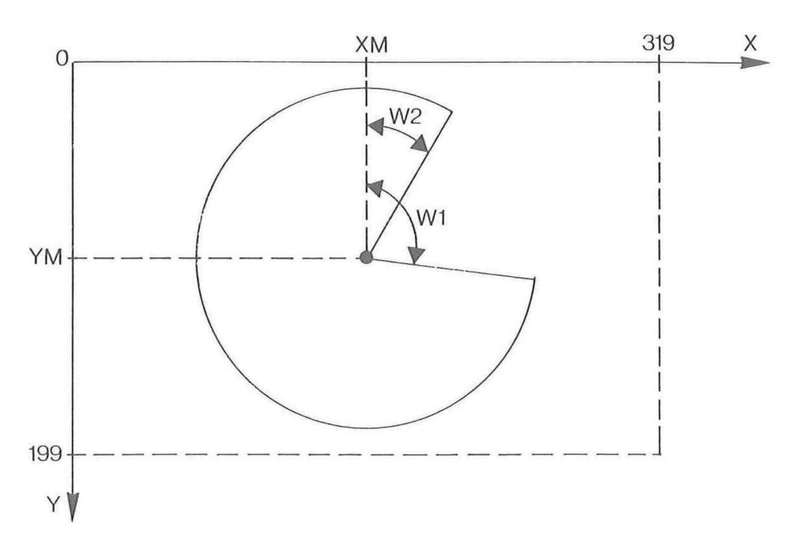


Bild 1.17: Zeichnen des Bogens, wenn W1 größer als W2 ist

Es wird der Bogen von W1 über 0 hinweg bis W2 gezeichnet.

Ein weiterer Parameter ist W, der Drehwinkel. Ebenso wie beim BOX-Befehl kann auch das durch CIRCLE erzeugte Objekt um den Mittelpunkt mit dem Uhrzeigersinn gedreht werden. Bild 1.18 zeigt die Verhältnisse dabei:

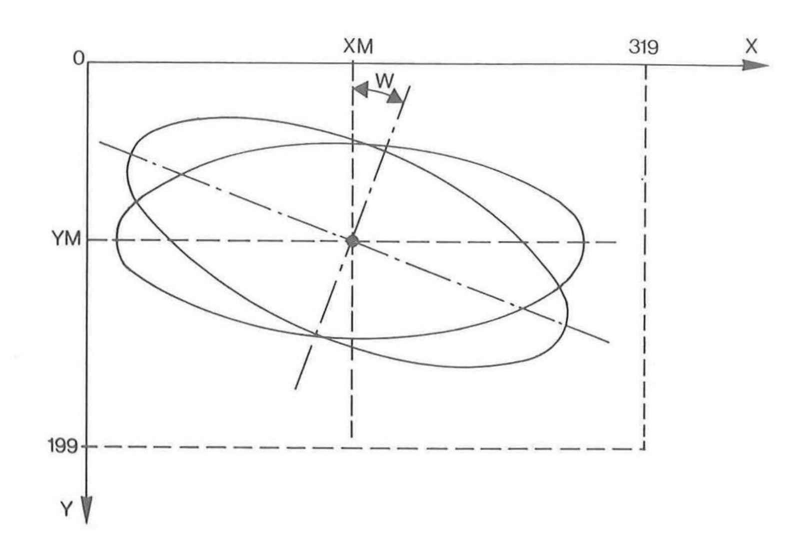


Bild 1.18: Drehung der Ellipse um den Winkel W um den Mittelpunkt XM,YM durch: CIRCLE1,XM,YM,RX,RY,,,W

Ohne zu murren nimmt der Computer Werte zwischen 0 und 65535 für W entgegen.

Der letzte Parameter S kann zwischen 1 und 255 groß sein. Er gibt an, nach welchem Winkel jeweils der nächste Kreis- oder Ellipsenpunkt zu berechnen ist. Die so ermittelten Punkte werden durch Geraden miteinander verbunden. Niedrige Werte von S bedeuten kleine Winkel. So ist der Wert S=2 voreingestellt, und die miteinander verbundenen Eckpunkte ergeben den Eindruck eines Kreises (oder einer Ellipse). Erhöht man aber S, dann beginnt der Kreis (ich erspare mir im weiteren den Hinweis auf die Ellipse) allmählich kantig zu werden. Man kann es zwar nicht erkennen, trotzdem ist der schnell gezeichnete Kreis (oder...) in Wahrheit ein 36-Eck, wenn S=10 gewählt wurde. Weitere Polygone ergeben sich zum Beispiel:

S=1	360-Eck	
S=2	180-Eck	
S=10	36-Eck	
S=20	18-Eck	
S=30	12-Eck	das kann man schon erkennen
S=60	6-Eck	
etc.		

Das Bild 1.19 soll dazu dienen, diese Angelegenheit zu illustrieren:

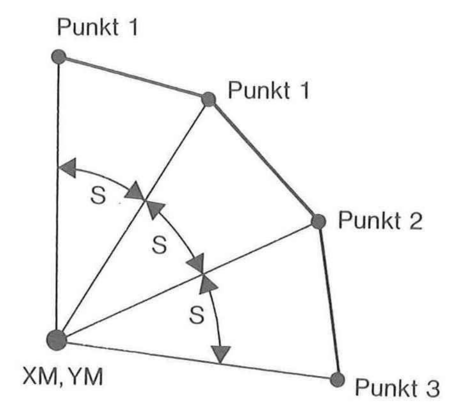


Bild 1.19: Der Parameter S bestimmt den Winkelabstand der berechneten Kreispunkte, zwischen denen Geraden gezogen werden.

Weil der gesamte Kreis einen Winkel von 360 Grad überstreicht, läßt sich eine Formel angeben, mit der gezielt n-Ecke erzeugt werden können:

$$S = 360/n$$

Wenn Sie also ein Dreieck (n=3) erzeugen möchten, müssen Sie S=120 eingeben. Die Formel zeigt uns noch zwei weitere Eigenschaften von S:

- Erstens kann man sie umdrehen und berechnen, wie viele Ecken ein Polygon hat bei einem bestimmten S-Wert:

$$n = 360/S$$

Dabei zeigt es sich, daß auch allerhand gebrochene Zahlen n zustande kommen. Weil es aber beispielsweise kein 7,2-Eck gibt (bei S=50), muß es sich hier um ein krummes 7-Eck handeln.

- Zweitens sind Eingaben größer als 180 sinnlos. S=180 führt zum "Zweieck", also zum Durchmesser des Kreises (siehe Bild 1.20):

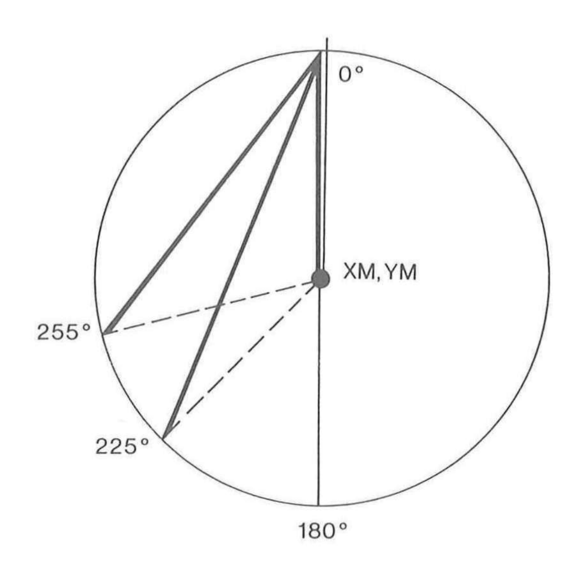


Bild 1.20: Wirkung von S-Werten ab 180. Gezeichnet wird jeweils die fette Linie.

Der Startpunkt ist immer oben (also bei einem Winkel von 0 Grad) zu finden, der Endpunkt liegt auf der Kreisperipherie um 180 Grad demgegenüber verdreht. Ein Weiterzählen um noch einmal 180 Grad führt wieder zum Startpunkt. Wächst S auf noch höhere Werte, dann wird nur noch eine Sehne gezeichnet vom Startpunkt zum Punkt 1, der beispielsweise bei S=225 um 225 Grad, bei S=255 um 255 Grad verdreht auftritt. Ein weiterer Punkt wird hier nicht mehr berechnet, weil damit der Ausgangspunkt bei 0 Grad überschritten würde.

Den Grafik-Cursor hätten wir fast noch vergessen: Der befindet sich (im Gegensatz zum BOX-Befehl) immer dort, wo der letzte Punkt des CIRCLE-Befehls gesetzt wurde.

Übrigens gilt es noch eine Besonderheit des CIRCLE-Befehls im Multicolor-Modus zu beachten. Gibt man hier nämlich einen Befehl zum Zeichnen eines Kreises in der Form

CIRCLE1,XM,YM,RX

ein, wobei ja automatisch RY = RX gesetzt wird, dann entsteht zwar auf dem Bildschirm ein Kreis. Der ist aber falsch, was Sie unschwer erkennen können, indem Sie oben und unten Tangenten anlegen. Das Ganze sähe beispielsweise so aus:

CIRCLE1,80,100,40:DRAW1,0,40TO159,40

DRAW1,0,120TO159,120

Als unverbildete Programmierer sollten wir erwarten, daß diese Geraden als Tangenten den Kreis berühren am oberen und am unteren Rand. Aber weit gefehlt! Sie schneiden ihn! Probieren wir dagegen einmal, den Kreis mittels

CIRCLE1,80,100,40,40

zu zeichnen, dann sollte ja dasselbe herauskommen. Aber oh Wunder: Eine Ellipse ergibt unser Werk. Woran liegt das? Die Ursache dafür liegt in der ungleichen Skalierung der X- und der Y-Achse im Multicolor-Modus. Wenn wir dem Computer den Auftrag erteilen, einen Kreis zu zeichnen, dann zeichnet er ihn - ohne Rücksicht auf Verluste oder Skalierungen oder ähnliche Feinheiten.

1.6 PAINT

Damit kann eine von Kurvenzügen umschlossene Fläche ausgefüllt werden. Sie muß aber auch wirklich völlig eingerahmt sein, denn das kleinste Loch in der Hülle führt dazu, daß auch die benachbarte Fläche bemalt wird. Im Befehl

PAINT A, X1,Y1, M

haben die einzelnen Größen folgende Bedeutung:

A ist - wie schon gehabt - wieder die Farbquelle, mit der das Ausmalen geschehen soll.

X1 und Y1 sind die Koordinaten eines Punktes, der in der auszufüllenden Fläche enthalten ist. Gleichzeitig ist das der Startpunkt der Malaktion, und hinterher liegt genau hier der Grafik-Cursor.

Ein etwas schwerer verständlicher Parameter ist der Modus M. Ist nämlich M gleich 0, dann wird so lange ausgefüllt, bis eine Umrahmung in der Farbe angetroffen wird, die im Parameter A gewählt wurde. Ist dagegen M gleich 1, dann zählt auch jede andere Farbe der Umrandung - sofern sie nur nicht die des Hintergrundes ist - als Grenze des Ausmalens. Befinden wir uns beispielsweise im Grafik-Modus3, dann passiert bei der Befehlskombination

BOX2,10,10,100,100:CIRCLE3,80,100,40: PAINT1,85,85

nicht das Ausmalen der "Schnittmenge" von Kreis und Rechteck, sondern ungerührt um unsere Bemühungen übermalt der Computer alles, was da auf dem Bildschirm zu sehen war. Der Grund ist, daß keine Umrahmung in der Farbe mit der Kennung A = 1 vorhanden war (BOX lief mit A=2 und CIRCLE mit A=3) und wir den Modus 0 gewählt haben. Modus 1 hätte unseren Wunsch erfüllt.

PAINT arbeitet im Modus M=1 nicht, wenn am Startpunkt schon eine andere Farbe als die Hintergrundfarbe gefunden wird. Im Modus M=0 dagegen wird immer dann ausgefüllt, wenn eine Farbe dort enthalten ist, die von der ausgewählten verschieden ist.

1.7 CHAR

Eigentlich kein Grafikbefehl im engeren Sinn ist dieser Befehl zum Drukken eines Textes. Seine interessanteste Anwendung ist aber die Beschriftung von Grafiken, was auf andere Weise (außer in den Splitscreen-Modi) nicht möglich ist.

CHAR A, XT,YT, A$, R

bildet einen Text A$ in der Farbe der Quelle A ab. Das erste Zeichen befindet sich bei den Koordinaten XT,YT. Das Koordinatensystem, das hier

eine Rolle spielt, ist das im Textmodus gültige (25 Zeilen zu je 40 Spalten, oder aber 80 Spalten). A$ kann jede beliebige Kombination von gültigen Zeichen enthalten, in den Grafikmodi werden allerdings die Steuerzeichen als grafische Zeichen mitgedruckt.

R ist eine Kennzahl, die es erlaubt, den Text normal oder invers darzustellen:

R = 0	Normaldarstellung
R = 1	Reversdarstellung

Sollte der Text nicht in einer Zeile Platz finden, dann wird er in der nächsten fortgesetzt, also ein Verhalten, wie wir es auch beim PRINT-Befehl finden.

Im Multicolormodus sehen die Zeichen häufig etwas merkwürdig aus. Die Zeichenmuster sind ja nicht zur Darstellung im Bitpaarmodus konstruiert. Soll ein Text in der Multicolorfarbe 2 gedruckt werden, dann ist A auf 0 und R auf 1 zu setzen. Soll dagegen die Multicolorfarbe 1 verwendet werden, hält man die Farbquelle A auf 1, und für R wählt man den Wert 0. So findet es sich jedenfalls im Handbuch (auf den Seiten 4-28 und 4-29). Zumindest mein Commodore 128 folgt diesem Rezept nicht, sondern es genügt, einfach A entsprechend der gewünschten Farbquelle (also 0,1,2,3) zu wählen.

Der Grafik-Cursor bleibt übrigens vom CHAR-Befehl unbeeindruckt.

1.8 LOCATE

Wir haben bei den bisher untersuchten Befehlen immer wieder einen Grafik-Cursor erwähnt und dessen Verhalten bei den verschiedenen Operationen. Sie haben diesen Cursor aber noch nie zu Gesicht bekommen. Sie werden das auch nie, denn er führt sein verborgenes Leben nur in einigen Speicherstellen, nie aber auf dem Bildschirm. Trotzdem ist er recht nützlich. Er dient gewissermaßen als Merkstelle für bestimmte Punkt-Koordinaten:

- als Startpunkt bei DRAW

  Beispielsweise für DRAW1 TO X2,Y2

- als Eckpunkt bei BOX

- als Mittelpunkt bei CIRCLE:

  Laut Handbuch soll man die Koordinaten XM,YM weglassen können, und dafür werde automatisch der Ort des Grafik-Cursors eingesetzt. Leider war das bei meinem Commodore 128 nicht nachvollziehbar. Es ergab sich nur allerlei Unsinn auf dem Bildschirm;

- als Startpunkt für PAINT:

  Auch hier kann man ohne Angabe der Startpunktkoordinaten Flächen ausmalen, denn es wird automatisch der Ort des Grafik-Cursors verwendet.

LOCATE X,Y führt diesen Grafik-Cursor an den durch die Koordinaten X,Y definierten Punkt.

1.9 RDOT

Wollen Sie erfahren, wo sich der Grafik-Cursor im Augenblick befindet, dann dient dieser Befehl Ihrem Wunsch. Das Argument von

RDOT(n)

hat folgende Bedeutungen:

n = 0	liefert die aktuelle X-Position
n = 1	ergibt die aktuelle Y-Position des Grafik-Cursors
n = 2	macht eine Aussage über die aktuelle Farbquelle

Die Koordinatenwerte, die RDOT-Anfragen ergeben, beziehen sich immer auf die grundlegenden Koordinatensysteme. Im hochauflösenden Grafik-Modus stammen die Werte daher immer aus einem System, dessen X-Werte zwischen 0 und 319 liegen und dessen Y-Werte von 0 bis 199 reichen. Im Multicolor-Modus geht die X-Achse nur bis 159. Das im Auge zu behalten, ist besonders für den nun folgenden Befehl wichtig.

1.10 SCALE

Mittels SCALE n, Xmax, Ymax können wir neue Koordinatensysteme auf unserem Bildschirm definieren. Die Kennzahl n kann zwei Optionen anwählen:

n = 0	Das reguläre System ist eingeschaltet: Grafikmodi 1 und 2: X von 0 bis 319, Y von 0 bis 199
n = 1	Ein verändertes System ist nun Bezugssystem geworden.

Wenn durch n=1 die neue Skalierung eingeschaltet wurde, gibt es zwei Möglichkeiten. Schreibt man lediglich

SCALE 1

dann hat man ein Koordinatensystem gewählt, dessen X- und Y-Koordinaten jeweils von 0 bis 1023 reichen. Der zweite Weg ist es, selbst die Reichweiten der Skalen festzulegen. Das geschieht durch die Angaben von Xmax und Ymax, den Höchstwerten auf der jeweiligen Achse. So wählt im Grafik-Modus 1 der Befehl

SCALE 1, 400, 300

ein System, dessen X-Achse von 0 bis 400, dessen Y-Achse von 0 bis 300 reicht (siehe Bild 1.21).

Die Werte Xmax müssen größer als 320 gewählt werden, die für Ymax größer als 200. Es ist so also nicht möglich, verkleinerte Systeme zu erzeugen. Im Gegensatz zu den Handbuch-Angaben, die nur Werte bis 1023 erlauben, ist es tatsächlich möglich, sowohl Xmax als auch Ymax bis 32767 anzugeben.

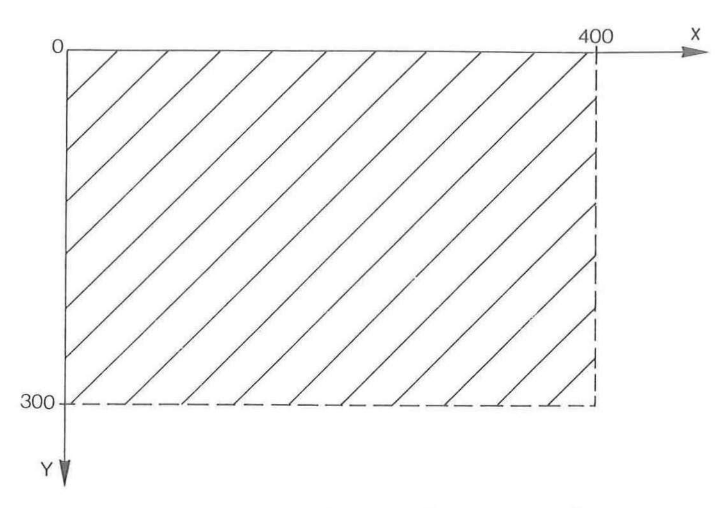


Bild 1.21: Wirkung von SCALE1,400,300 in den Grafik-Modi 1 und 2

Im Multicolormodus ist zu beachten, daß weiterhin auf der X-Achse immer Bit-Paare dargestellt werden. Im Grafik-Modus 3 oder 4 erzeugt der Befehl

SCALE 1, 400, 300

ein System, dessen X-Koordinaten von 0 bis 200 und dessen Y-Koordinaten von 0 bis 300 reichen (siehe Bild 1.22):

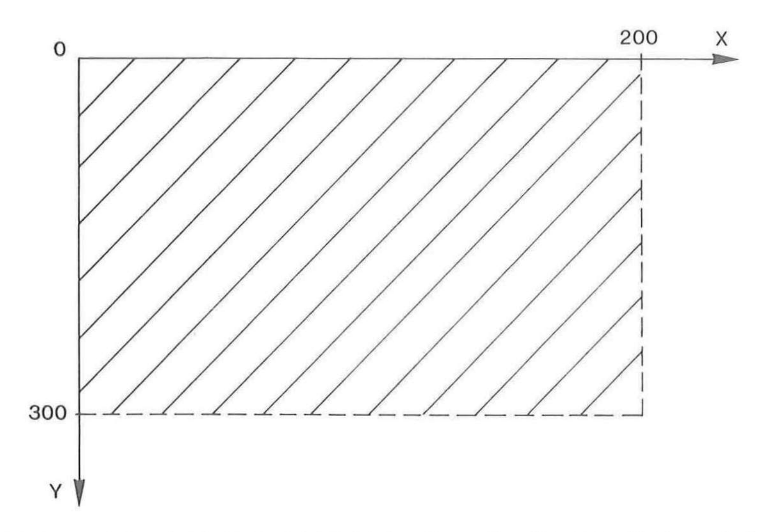


Bild 1.22: So wirkt SCALE1,400,300 in den Grafik-Modi 3 und 4

Natürlich werden weiterhin die Abbildungen auf dem Bildschirm mit einer Auflösung von insgesamt 64000 Punkten (bzw. 32000 im Multicolormodus) dargestellt. Die Bit-Map bleibt unverändert. Lediglich der Berechnungsweg zur Darstellung der Punkte verändert sich. Etwas unglücklich werden viele Anwender des Commodore 128 darüber sein, daß der SCALE-Befehl keine weitergehenden Umstellungen des Koordinatensystems ermöglicht. Wünschenswert wäre sicherlich die Möglichkeit gewesen, den Ursprung zu verschieben (wie es der TRS-Befehl der Grafikerweiterung HIRES-3 für den C64-Modus erlaubt). Durch diese mindere Ausstattung kommt dem SCALE-Befehl nicht die Bedeutung zu, die er haben könnte, denn jeder Anwender muß die langwierigen Transformationen selber programmieren.

SCALE hat auch Einfluß auf das Ergebnis eines CIRCLE-Befehls. Ebenso, wie wir es vorhin zum Multicolor-Modus beobachten konnten, wo ohne Rücksicht auf den anderen Maßstab der Y-Achse ein Kreis gezeichnet wurde, findet das hier in allen Grafik-Modi statt. Wenn also der richtige Kreis - das heißt derjenige, der bei allen Maßstäben in Y-Richtung den gleichen Durchmesser aufweist wie in X-Richtung - gezeichnet werden soll, muß das nun immer durch den CIRCLE-Befehl mit zwei Halbachsenangaben geschehen. Das sollen Ihnen die Bilder 1.23 und 1.24 erläutern:

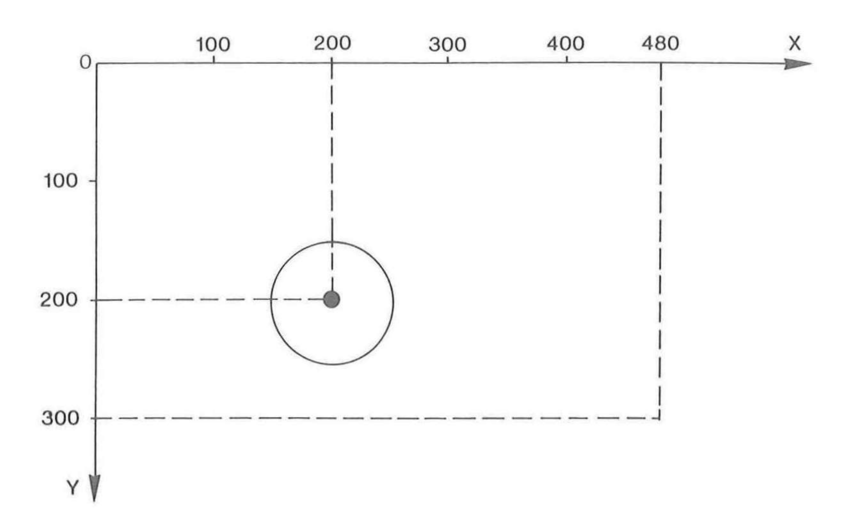


Bild 1.23: SCALE1,480,300 erzeugt in den Grafik-Modi 1 und 2 dasselbe Achsenverhältnis wie das normale System:
320/200 = 480/300 = 1,6
Der Kreis CICRLE1,200,200,50 ist identische mit dem Kreis CIRCLE1,200,200,50,50.

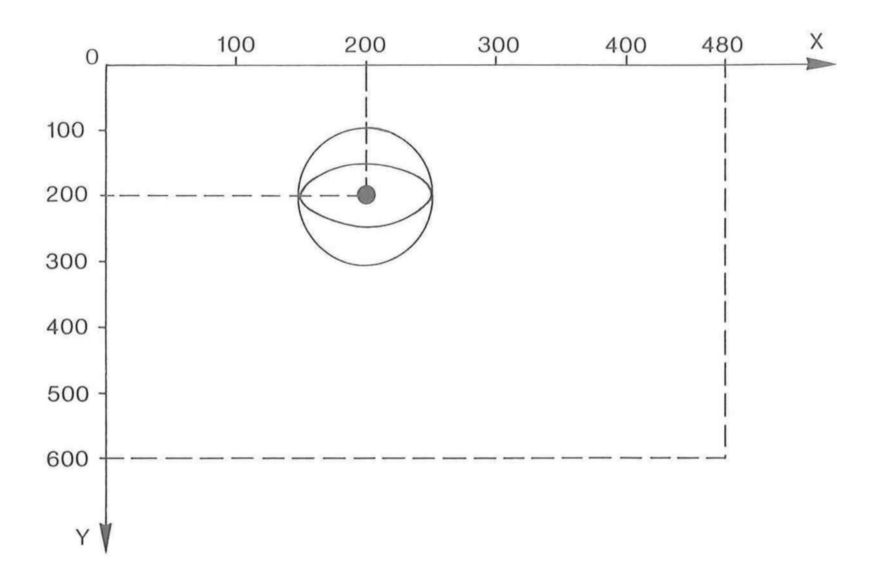


Bild 1.24: Durch SCALE1,480,600 ist ein Achsenverhältnis von 480/600 = 0,8 geschaffen worden. CIRCLE1,200,200,50 erzeugt den falschen Kreis, CIRCLE1,200,200,50,50 zeichnet zwar den richtigen Kreis, der aber zur Ellips verzerrt wird.

Die Verhältnisse in diesen Abbildungen beziehen sich auf die Grafikmodi 1 und 2. Das Achsenverhältnis im Multicolormodus ist durch den Faktor 2 in X-Richtung verändert. Man sollte also festhalten, daß immer dann unverzerrte Bilder zustande kommen, wenn die Achsenverhältnisse wie folgt sind:

Grafikmodi 1 und 2:X:Y = 1.6
Grafikmodi 3 und 4:X:Y = 3.2

Der SCALE-Befehl kann - nebenbei bemerkt - innerhalb eines Programms oder bei ein und demselben Bildschirm - mehrfach angewendet werden. Das ergibt interessante Effekte.

1.11 SCNCLR

"SCreeN CLeaR" ist englisch und heißt etwa "Bildschirm freimachen". Das bedeutet, daß im 40-Zeichen-Textmodus der Speicherbereich ab $400, in den Grafikmodi sowohl die Bit-Map als auch das Farb-RAM ab $1C00 gelöscht bzw. mit Leerzeichen oder Farbcode beschrieben werden. Welche Bereiche gelöscht werden, welche Bildschirme also, entscheidet das Argument von

SCNCLR n

Die Zuordnungen von n sind hier die gleichen, die auch beim GRAPHIC-Befehl verwendet werden.

Ein gerade sichtbarer Grafik-Bildschirm ist auch löschbar ohne Angabe eines Argumentes, SCNCLR genügt in diesem Fall. Sollte im Textmodus ein Fenster mittels WINDOW definiert sein, dann bezieht sich der Löschbefehl auf den Fensterinhalt.

Bei der Anwendung des SCNCLR-Befehls in den Grafik-Modi muß bedacht werden, daß für den Multicolor-Modus und für den hochauflösenden Modus dieselbe Bit-Map benutzt wird. Nach SCNCLR3 ist also auch ein im Grafik-Modus 1 sinnvolles Bild verschwunden.

1.12 WIDTH

"Width" ergibt aus dem Englischen ins Deutsche übersetzt den Begriff "Breite". Gemeint ist damit die Strichstärke, in der Zeichenbefehle auf dem Bildschirm realisiert werden.

Mit WIDTH 1	schaltet man die normale Strichdicke ein,
mit WIDTH 2	wird allen grafischen Objekten die doppelte Strichbreite zugeordnet.

Alle Zeichenbefehle nach einem WIDTH-Befehl führen dann zur darin angegebenen Strichstärke.

1.13 RCLR

Dieser Befehl bildet das Gegenstück zum COLOR-Befehl. Mittels

RCLR n

ist es möglich festzustellen, welche Farbzuordnungen zu den verschiedenen Quellen getroffen worden sind. n ist dabei eine Kennzahl mit derselben Bedeutung wie der erste Parameter der COLOR-Anweisung.

PRINT RCLR(2)

gibt also Auskunft über die Farbe, die als Multicolorfarbe 1 definiert wurde.

1.14 RGR

Vermutlich relativ selten benötigt, ergibt

PRINT RGR(x)

den aktuellen Grafik-Modus. Das Argument x ist hier lediglich ein Dummy, hat also keine Funktion außer der eines Platzhalters. Die sich ergebende Zahl entspricht dem ersten Argument des GRAPHIC-Befehls, gibt also den Grafik-Modus an.

1.15 Noch einmal Farbwechsel

Buchstäblich in letzter Minute flattert mir ein ROM-Listing auf den Tisch. Damit Sie auch noch etwas davon haben, soll hier gleich ein weiterer - recht bequemer - Weg vorgestellt werden, auf dem der Wechsel der Farben von hochauflösenden und von Multicolorbildern bewältigt werden kann.

Zur Erinnerung: Probleme gab es vor allem mit dem Neudefinieren der Multicolorfarbe 2. Wir hatten zwar festgestellt, daß sie in der Speicherstelle $85 (=dez.133) als Farbcode-1 gespeichert wurde, wußten uns aber nur mit einem Trick zu helfen, nämlich durch Abspeichern und Neuladen des Bildes mit neuer Multicolorfarbe. Diese Zeiten sind nun vorbei. Künftig können Sie diese Zeile dazu verwenden:

```
POKE133,X:BANK15:SYS DEC("6B17")
```

Das geht blitzschnell, und dabei wird eine Interpreter-Routine zum Neubelegen des Farb-RAM bei $D800 (mit gelöschtem Bit 0 der Speicherstelle $01) verwendet.

Einen ähnlichen Weg können wir auch zum Neubelegen des Farb-RAM ab $1C00 begehen, der ja sowohl im hochauflösenden als auch im Multicolormodus eine Rolle spielt. In der nun vorgestellten kurzen Routine spielen die beim COLOR-Befehl schon vorgestellten Speicherzellen

$03E2hochauflösende
$03E3Multicolorgrafik

die Hauptrolle. Deren Inhalt wird verwendet - als Variable F - beim SYS-Befehl. Dabei ergibt sich F auf die Weise, die wir schon im Abschnitt über den COLOR-Befehl hergeleitet hatten:

F = 16*(F1-1)+(F2-1)

Hier ist dann

F1 im hochauflösenden Modus die Farbe des Vordergrundes und im Multicolor-Modus ebenfalls die Vordergrundfarbe, also die mit der Bitkombination 01.

F2 ist im hochauflösenden Modus die Hintergrundfarbe und in den Grafik-Modi 3 und 4 die Multicolorfarbe 1 (Bitkombination 10).

Mit dem so gewonnenen F schreiben wir nun:

BANK15:SYS DEC("6B43"),F

Damit haben wir nun alle Farbänderungen voll im Griff und können jederzeit einem Bild ein neues Gesicht verleihen.

2 Sprites & Shapes

Um die Programmierung der Sprites auf dem C64 zu erklären, ist schon viel Druckerschwärze verbraucht worden. Fast schien es eher die Aufgabe von Assembler-Programmierern zu sein, durch die POKE-Wüsten und Zahlensysteme zu gelangen, um schließlich das Ziel - ein bewegtes Sprite auf dem Bildschirm - zu erreichen. Im Commodore 128-Modus erspart uns das BASIC 7.0 all diese Strapazen: Sprites erstellen und zu verwalten wird zum Vergnügen. Nachfolgend lernen wir die dazu nötigen Befehle bis in ihre Feinheiten kennen. Außerdem wird uns eine neue Gruppe von Grafikobjekten beschäftigen: die Shapes.

2.1 SPRITES

Sollten sie Ihnen noch nicht begegnet sein, die kleinen Kobolde ("sprite" ist englisch und heißt auf deutsch "Kobold" oder "Gespenst"), dann seien sie hiermit kurz vorgestellt: Es handelt sich um bewegliche grafische Objekte (daher auch der Ausdruck MOB=movable objects, der manchmal verwendet wird), die sowohl als mehr- als auch als einfarbige (dann hochaufgelöste) Figuren ohne Rücksicht auf den gerade eingeschalteten Grafikmodus munter über Text- und Grafikbildschirm huschen können. Die gesamte Verwaltung der Sprites geschieht durch den VIC-Chip, was ihr Auftreten auf den 40-Zeichen-Bildschirm beschränkt.

2.1.1 Die Befehle zur Spriteprogrammierung

10 BASIC-Befehle dienen zum Erstellen und Verwalten von Sprites:

2.1.1.1 SPRDEF

Damit ruft man den Sprite-Editor auf. Der Bildschirm wird gelöscht, links oben erscheint ein Raster mit 24 mal 21 möglichen Positionen, darunter fragt der Computer nach der gewünschten Sprite-Nummer. Wir haben nun die Wahl zwischen 8 vorgesehenen Sprites. Sollten Sie schon ein MOB im Speicher haben, dann rufen Sie es nun mit seiner Nummer auf, ansonsten müssen Sie sich jetzt festlegen: Alle weiteren Sprite-Befehle beziehen sich immer auf diese Nummer. Nach deren Eingabe wird das Rasterfeld mit dem Inhalt eines speziellen Speicherbereiches (aus $0E00 bis 0FFF = dezimal 3584 bis 4095) gefüllt, der die Informationen des gewählten Sprites enthält. Falls noch kein Sprite definiert war, kann dabei auch allerlei Byte-Müll abgebildet werden, der uns aber vorläufig nicht erschüttern soll. In der linken oberen Ecke des Rasterbereiches meldet sich ein Kreuz: Das ist der Sprite-Cursor. Oben rechts neben dem Raster ist unser Sprite so abgebildet, wie es im Normalbetrieb später auf dem Bildschirm erscheinen wird.

Bevor wir auf die einzelnen Optionen des Sprite-Editors eingehen, noch eine Bemerkung. Alles bisher Beschriebene benutzt den Grafik-Bildschirm. Ebenso wie bei der Verwendung der hochauflösenden und der Mehrfarbengrafik wird dazu der BASIC-Anfang nach $4000 verschoben, also über die Bit-Map.

Sehen wir uns nun die Möglichkeiten unseres Editors an:

CLR/HOME	Damit kann das Definitionsfeld gelöscht werden. Das befreit uns vom Byte-Müll.
HOME	Der Sprite-Cursor marschiert in die linke obere Ecke.

Cursor-Tasten

Damit können wir den Sprite-Cursor über das Rasterfeld wandern lassen.

RETURN — Der Cursor wird auf den Anfang der nächsten Zeile gesetzt.

A — Die Wiederholungsfunktion der Tasten 1 bis 4 kann durch einmaliges Drücken aus-, durch nochmaliges Drücken wieder eingeschaltet werden.

M — Einmal Drücken schaltet den Mehrfarbenmodus ein. Dadurch bekommen die Tasten 1 bis 4 eine neue Bedeutung.

Nochmaliges Drücken schaltet wieder auf den normalen Hochauflösungsmodus.

X — Ein Druck verdoppelt das Sprite in waagerechter Richtung, ein weiterer Tastendruck erzeugt wieder ein Sprite in normaler Breite.

Y — Mit dieser Taste passiert in der Senkrechten dasselbe wie in der Horizontalen bei der X-Taste.

Zifferntasten 1 bis 4

a) Im Normalmodus setzt

1 die aktuelle Hintergrundfarbe (0),
2-4 die aktuelle Vordergrundfarbe (1).

b) Im Mehrfarbenmodus setzt

1 die Hintergrundfarbe (00),
2 eine Multicolorfarbe (01),
3 die aktuelle Vordergrundfarbe (10),
4 die andere Multicolorfarbe (11).

Zur Erläuterung:

- Im Normalmodus tritt jedes gesetzte Bit (=1) als Bildpunkt in der Vordergrundfarbe auf, alle nicht gesetzten (=0) erscheinen in der Hintergrundfarbe.
- Im Mehrfarbenmodus tragen die Bit paarweise zur Farbgebung bei. Deshalb ist hier der Sprite-Cursor auch doppelt so breit.

Control 1-8 wählt die aktuellen Vordergrundfarben 1-8 aus.
Commodore 1-8 leistet dasselbe mit den Farbcodes 9 bis 16.

Zwei Optionen, die recht nützlich sind, werden merkwürdigerweise im Handbuch nicht erwähnt:

C — Das ist eine Kopierfunktion. Nach Tastendruck erscheint die Frage "Copy from?". Nach Eingabe einer Spritenummer wird das dazugehörige Muster in das aktuelle Sprite kopiert und dieses dabei überschrieben. Das ist ganz sinnvoll, wenn man mehrere ähnliche Sprites erzeugen möchte.

Dasselbe passiert auch, wenn man die Funktionstaste F1 betätigt.

Ist man einmal versehentlich auf diese Tasten geraten, dann kann man durch RETURN wieder in den normalen Editorzustand gelangen.

Control C — erlaubt das Umschalten zwischen den Sprites. Nach Betätigen dieser Tastenkombination verschwindet die aktuelle Spritenummer. Gibt man nun die gewünschte Nummer ein, schaltet sich der dazugehörige Sprite an.

Zwei Möglichkeiten zum Verlassen des Sprite-Editors sind vorgesehen:

STOP-Taste — Nach dem Betätigen der Taste verliert man alle neuen Eingaben im Definitionsfeld.

Gibt man anschließend eine Spritenummer ein, wird das dazugehörige Muster eingeladen.

Ein anschließendes RETURN führt zum Verlassen des Editors. Mit READY meldet sich der Textbildschirm zurück.

Shift-RETURN Abspeichern des aktuellen Sprite im Bereich $0E00 - 0FFF.

Ebenso wie bei der STOP-Taste kann man danach durch Eingabe einer neuen Spritenummer das nächste Sprite zur Bearbeitung in den Raster holen.

Ein RETURN führt zum Verlassen des Sprite-Editors.

Interessant an SPRDEF ist, daß man diesen Befehl auch im Programm-Modus verwenden kann. Auf diese Weise kann man im Programm ein oder mehrere Sprites erstellen, und anschließend - also nach Aussteigen mit STOP/RETURN oder Shift-Return/RETURN - läuft das Programm weiter.

Das also sind alle Möglichkeiten unseres Sprite-Editors: Leider fehlen einige nötige Optionen. So ist es nicht möglich, die frisch zusammengebauten Sprites in DATA-Zeilen abzulegen, die Multicolorregister müssen vor dem SPRDEF-Aufruf belegt werden, die Muster kann man nicht invertieren, drehen oder spiegeln.

2.1.1.2 SPRCOLOR

Im Normalmodus kann die Punktfarbe eines Sprites und auch die Farbe der Bitkombination 10 des Multicolormodus durch den SPRITE-Befehl bestimmt werden oder in SPRDEF durch die Kombination von Control- oder Commodoretaste mit einer Ziffer. Die Belegung der Multicolorregister mit Farbe - das entspricht den SPRDEF-Zifferntasten 2 und 4 -, also der Bitkombinationen 01 und 11, erfolgt durch den SPRCOLOR-Befehl. Die Verwendung von

SPRCOLOR A,B

führt

in Multicolorregister 1 (Bit-Kombination 01 oder SPRDEF-Taste 2) zur Farbe A,

in Multicolorregister 2 (Bit-Kombination 11 oder SPRDEF-Taste 4) zur Farbe B.

Somit ergibt der Befehl

SPRCOLOR 3,6

die Farbe ROT der Bitkombination 01 und GRÜN der Kombination 11.

2.1.1.3 SPRITE

Dies ist zweifellos der wichtigste Befehl für ein schon vorhandenes Spritemuster. Er schaltet ein Sprite ein und legt seine Eigenschaften fest. Syntax:

SPRITE A,B,C,D,E,F,G

Keine Angst vor den vielen Parametern: Sie müssen nicht immer alle angeben. Zur Bedeutung der einzelnen Buchstaben:

A	Spritenummer (1 - 8)
B	0 = Ausschalten des Sprites 1 = Einschalten des Sprites
C	Farbcode für die Vordergrundfarbe (das ist das Bitpaar 10 bei Multicolorsprites) (1 bis 16)
D	Priorität gegenüber Text: 0 = Sprite vor Text 1 = Text vor Sprite Nebenbei: Die Priorität von Sprites untereinander wird durch die Sprite-Nummer geregelt: 1 vor 2 vor 3 ... vor 8.
E	X-Vergrößerung: 0 = normale Größe 1 = doppelte Größe
F	Y-Vergrößerung: 0 = normale Größe 1 = doppelte Größe

Sind sowohl E als auch F auf 1 gesetzt, dann erhält man ein 4fach ausgedehntes Sprite.

G Modus:
0 = normales Hires-Sprite
1 = Multicolor-Sprite

Ein Beispiel soll die Wirkung des Befehls demonstrieren:

SPRITE 1,1,3,0,1,1,0

Dadurch wird Sprite 1 eingeschaltet. Es ist rot, erscheint vor Bildschirmzeichen und ist sowohl in horizontaler wie auch in vertikaler Richtung verdoppelt. Es handelt sich um ein normales (also Hires-Sprite). Nun soll das gleiche Sprite auf die normale Größe gebracht werden:

SPRITE 1,,,,0,0

Es genügt, die Werte anzugeben, die zu ändern sind. Alle anderen werden so übernommen, wie sie vorher festgelegt wurden. Die Spritenummer und die Kommas vor dem zu ändernden Wert sind obligatorisch.

2.1.1.4 MOVSPR

Das ist ein recht vielgestaltiger Geselle. MOVSPR kommt von "move sprite", also hat dieser Befehl mit der Positionierung und Bewegung von MOBs zu tun. Drei Varianten sind möglich:

a) MOVSPR n,X,Y

Das Sprite mit der Nummer n wird an den Ort mit den Koordinaten X,Y gesetzt. Als Bezugspunkt beim Sprite dient die linke obere Ecke (auch wenn diese unsichtbar ist). Bild 2.1 zeigt Ihnen das normale Sprite-Koordinatensystem, das in allen Grafik-Modi gültig ist.

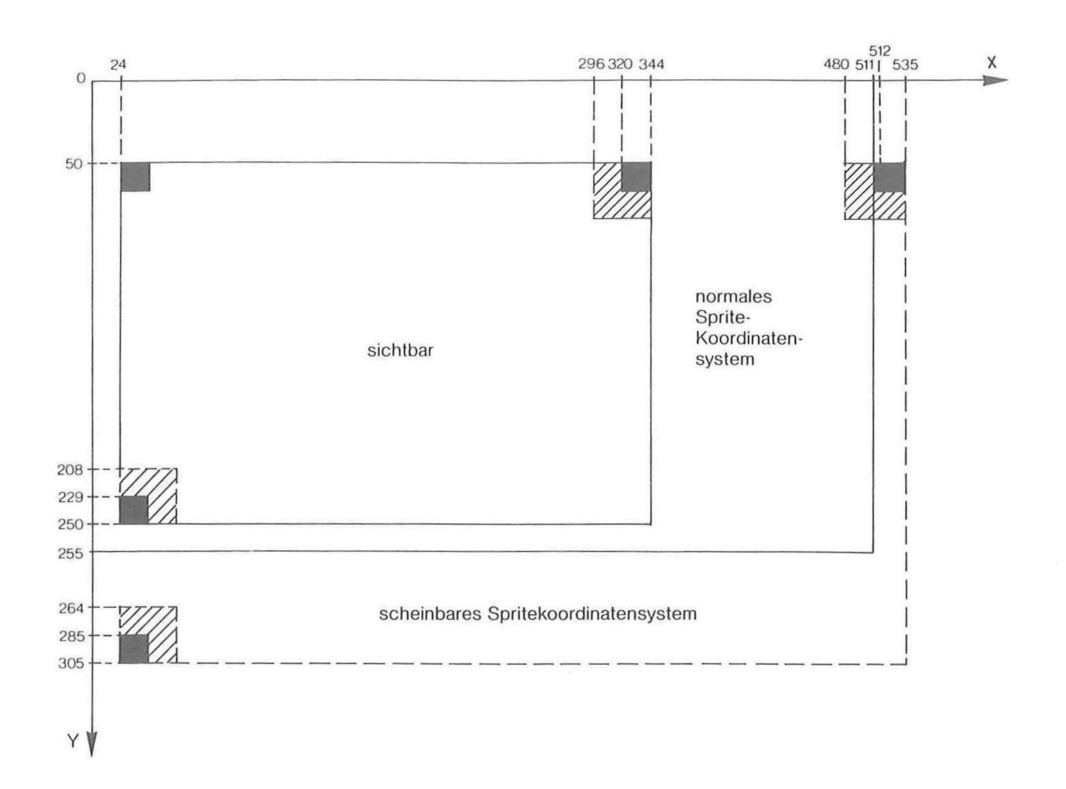


Bild 2.1: Das normale Sprite-Koordinatensystem

Der sichtbare Bildschirm erstreckt sich von X=24 bis 344, und in der Vertikalen reicht er von 50 bis 250. Für X und Y dürfen Werte zwischen 0 und 65535 eingesetzt werden. Bild 1 skizziert auch, bei welchen Koordinatenangaben ein Sprite voll sichtbar ist:

Links oben müssen dazu die Koordinaten (24/50) gewählt werden. Ein normales Sprite ist bei X=320, ein in X-Richtung gedehntes bei X= 296 gerade noch voll sichtbar. Entsprechend gilt für die Y-Werte. Normales Sprite bis Y=229, in Y-Richtung gedehntes bis Y=208.

Größere Koordinatenwerte führen zum Herauswandern aus dem sichtbaren Bereich. überschreiten die X- oder Y-Werte eine bestimmte Grenze, dann treten die MOBs auf der jeweils gegenüberliegenden Seite wieder ins Bild. Das ist ab X=513 (bei gedehnten Sprites ab 481) und ab Y=286 (bei gedehnten Sprites ab 265) der Fall. Eine weitere Erhöhung der Koordinaten

führt dann zum Wandern über den Bildschirm, zum erneuten Verschwinden und schließlich Wiederauftauchen auf der anderen Seite und so weiter.

Vorhin war die Rede vom "normalen" Sprite-Koordinatensystem. Daraus kann messerscharf geschlossen werden, daß es auch noch ein anderes gibt, ein "nicht normales". Das ist tatsächlich der Fall, und es handelt sich nicht nur um eines, sondern um eine ganze Menge verschiedener Systeme. Haben Sie nämlich durch den SCALE-Befehl ein neues Hires- oder Multicolor-Bildschirmsystem definiert, dann beziehen sich X und Y auf dieses, was manchmal zu einiger Verwirrung führen kann. Sollte also die Gefahr bestehen, daß ungewollt aus einer früheren Programmphase ein anderes Bildschirmsystem gültig ist, dann kann durch SCALE 0 der normale Zustand hergestellt werden.

Hier noch ein Beispiel:

MOVSPR 1,300,100

setzt Sprite 1 an die Stelle (300/100) im jeweils gültigen Koordinatensystem.

b) MOVSPR n,+/-X,+/-Y

Damit kann man Sprite n relativ um + oder -X (und/oder Y) zur aktuellen Position verschieben. X und Y dürfen wieder Werte bis 65535 annehmen, wobei durch das Vorzeichen nun sogar eine Skalenbreite von -65535 bis +65535 möglich wird. Auch hier beziehen sich X und Y auf das jeweils durch SCALE definierte Koordinatensystem (ohne SCALE-Anwendung also auf das normale Sprite-System aus Bild 1).

Auf diese Weise kann - beispielsweise in einer Schleife - das Sprite praktisch endlos über den Bildschirm wandern. Es tritt nämlich keine Fehlermeldung auf, wenn die Summe aus alter Position und Verschiebung größer als 65535 wird. Beispielsweise ist es ohne weiteres möglich (ob es sinnvoll ist, wäre eine andere Frage), folgende Kombination zu verwenden:

MOVSPR 8,100,65000:MOVSPR 8,100,+65000

Auch ein Unterlauf unter Null ist auf diese Weise erlaubt.

Wie Sie aus dem Beispiel erkennen können, kann die MOVSPR-Anweisung auch kombiniert verwendet werden:

MOVSPR 1,+10,100 verschiebt Sprite 1 um +10 Einheiten relativ zum vorangegangenen X-Wert und auf die Y-Koordinate 100.

MOVSPR 2,50,-20 läßt Sprite 2 auf die X-Koordinate 50 wandern und gleichzeitig um 20 Y-Einheiten nach oben.

c) MOVSPR n,W#V

Diese Variante des Spritebewegungs-Befehls ist eine feine Sache: Unabhängig vom sonstigen Programmgeschehen wird das Sprite mit das Nummer n mit einem Richtungswinkel W und der Geschwindigkeit V über den Bildschirm gesteuert. Der Winkel W wird in Grad angegeben. Bild 2.2 zeigt die möglichen Richtungen, die der Kompaßrose entsprechen:

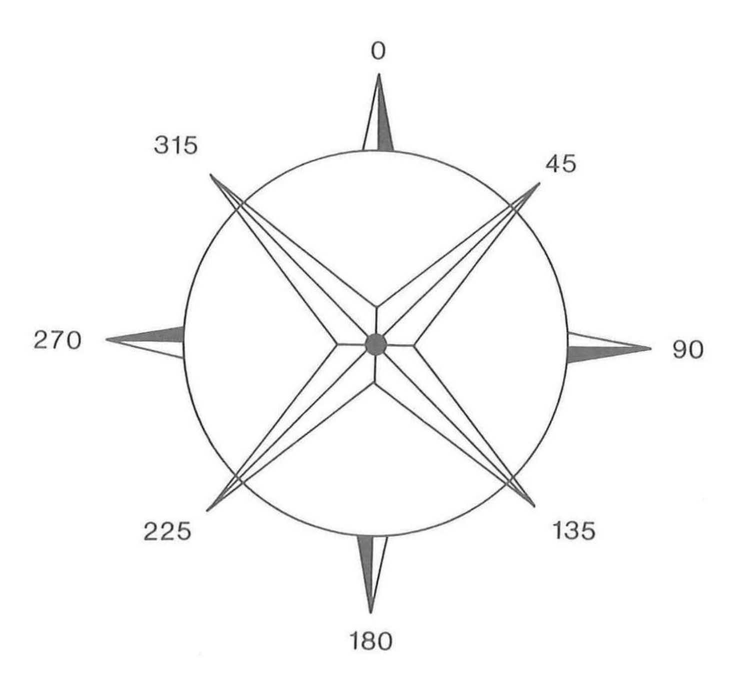


Bild 2.2: Die Kompaßrose: MOVSPR

Ohne Fehlermeldung sind Eingaben möglich zwischen -65535 und +65535. Sinnvoll scheint das aber allenfalls in einer Schleife, die ständige Richtungswechsel durchführt, zumal negative Werte und Angaben größer als 33024 keine vernünftige Reaktion ergeben.

Auch für die Geschwindigkeit V sind Angaben zwischen -65535 und 65535 möglich. Als sinnvoll erweisen sich hier nur Werte zwischen 0 und 15. 0 stoppt die Bewegung, und 15 ist die höchste erreichbare Geschwindigkeit. Für höhere V-Werte ergibt sich die Geschwindigkeit dann zu modulo(16). (Damit ist der Rest gemeint, der bei Division der eingegebenen Zahl durch 16 verbleibt. So ergibt V=30 dieselbe Schnelligkeit wie V=14.)

Eine Eigenart dieser MOVSPR-Variante ist es, daß die Wirkung dieses Befehls anhält, bis die Bewegung ausdrücklich durch MOVSPR n,W#0 auf Null gesetzt wird. Auch ein abgeschaltetes Sprite wandert weiter, was Sie leicht mal ausprobieren können:

Der Befehl

MOVSPR 1,90#15

läßt Sprite 1 mit Höchstgeschwindigkeit horizontal nach rechts über den Bildschirm jagen.

2.1.1.5 SPRSAV

Das ist ein sehr interessanter Befehl, dessen Anwendung wir später noch detailliert untersuchen werden. Zwei mögliche Varianten sind vorgesehen:

a) SPRSAV n,A$

Einem String (hier also A$) wird das Bitmuster des Sprites n zugeordnet. Mittels

SPRSAV 1,A$(2)

ordnet man das Bitmuster des Sprites mit der Nummer 1 dem Stringarrayelement A$(2) zu.

b) SPRSAV A$,n

Das ist der umgekehrte Weg: Das in A$ enthaltene Bitmuster definiert nun das Sprite mit der Nummer n.

Durch SPRSAV A$(2),8 wird das in A$(2) gespeicherte Muster in das Sprite Nummer 8 gelesen.

Anstelle von A$ kann jede Stringvariable - auch ein Array-Element - verwendet werden. Einige Möglichkeiten, die auf diese Weise gegeben sind: "Klonen" eines Sprites, Definieren von mehr als 8 Sprites (jeweils 8 sind gleichzeitig auf dem Bildschirm aktivierbar), schneller Austausch von Spritemustern.

2.1.1.6 COLLISION

Was im C64-Modus dem Assembler-Programmierer vorbehalten bleibt oder nur durch aufwendige POKE- und PEEK-Operationen realisiert werden kann, ist im Commodore 128-Modus mit diesem und dem folgenden Befehl möglich:

COLLISION A,NNNN

fängt drei Typen von Ereignissen ab und setzt die Programmbearbeitung in Zeile NNNN fort. A ist dabei eine Typkennung:

1 Sprite/Sprite-Kollision
2 Sprite/Text-Kollision
3 Lichtgriffelaktivierung

Das Serviceprogramm ab NNNN ist wie ein BASIC-Unterprogramm zu behandeln, also auch durch RETURN abzuschließen. Die weitere Bearbeitung geschieht dann an der Stelle, an der bei dem auslösenden Ereignis unterbrochen worden ist.

COLLISION stellt lediglich fest, welcher Typ stattgefunden hat. Bei Spritezusammenstößen kann also damit nicht die Spritenummer identifiziert werden. Es ist erlaubt, mehrere Kollisionstypen gleichzeitig zu aktivieren. Eine Aktivierung innerhalb eines Unterbrechungs-Service-Unterprogramms ist nicht möglich: Es findet zu einer Zeit nur die Bearbeitung eines Typs statt. Falls also mehrere aktiviert sind, sollte einer der ersten Befehle im Un-

terprogramm das Abschalten aller Kollisionsabfragen sein mittels COLLISION A (ohne Zeilennummer). Am Ende der Routine kann dann COLLISION A,NNNN wieder eingeschaltet werden.

Nicht immer ist es sinnvoll, so zu verfahren (also die Kollisionsabfrage am Ende der Routine wieder einzuschalten): Falls beispielsweise der Typ 1 (Sprite/Sprite-Zusammenstoß) als Auslöser definiert ist, muß bedacht werden, daß die Überlappung zweier Sprites einige Zeit andauert und deshalb das Unterprogramm mehrfach angesteuert wird. Im beigefügten Testprogramm 1 ist dieser Effekt deutlich an den vielen Textwiederholungen zu erkennen.

Testprogramm 1

Kollision von Sprites

```
10 REM ***** TESTPROGRAMM 1 - COLLISION *****
20 BLOAD"SPRITES2",ONB0
30 SPRCOLOR3,6
40 MOVSPR7,300,100:MOVSPR8,300,200
50 SPRITE7,1,2,0,0,0,1:SPRITE8,1,1,1,1,1,1
60 COLLISION1,90
70 MOVSPR8,+0,+5
80 GOTO70
90 COLLISION1
100 PRINT"KOLLISION ERFOLGT"
110 COLLISION1,130
120 RETURN
130 COLLISION1
140 PRINT"NOCH EINE KOLLISION"
150 COLLISION1,130
160 RETURN
```

Unter welchen Umständen wird eine Kollision erkannt? Immer dann, wenn sich sichtbare Teile von Sprites gegenseitig oder mit Bildschirmobjekten (bei Typ 2) überlagern. Abgeschaltete Sprites werden nicht berücksichtigt, wohl aber Zusammenstöße außerhalb des sichtbaren Bereiches (das Handbuch behauptet das Gegenteil).

Probieren Sie doch einmal in Zeile 40 die Spritepositionierung:

40 MOVSPR 7,345,100:MOVSPR 8,345,200

Nach dem Start ist keines der beiden Sprites mehr auf dem Bildschirm zu sehen, Kollisionen werden aber gemeldet.

Zwei Aspekte verdienen noch Erwähnung: Zum einen darf man ein Programm, in dem der COLLISION-Befehl verwendet wird, nicht durch STOP unterbrechen. Tut man es trotzdem, dann funktioniert nach einem CONT die Kollisionsabfrage nicht mehr. Zum anderen: Natürlich können alle Sprite-Befehle auch ohne 40-Zeichen-Bildschirm betrieben werden, man sieht nur nichts. Das kann bei COLLISION ganz rätselhafte Effekte erzeugen. So wären Spiele denkbar, die ein Sprite per Joystick durch Hindernisse hindurchbewegen. Das Ganze geschieht im Dunkel (also mit abgeschaltetem oder nicht vorhandenem 40-Zeichen-Bildschirm) und lediglich die Reaktion auf eine Kollision erfolgt auf dem 80-Zeichen-Bildschirm. Der Phantasie sind keine Grenzen gesetzt.

2.1.1.7 BUMP

Der Name sagt's bereits: Auch hier geht es um Zusammenstöße. Mit dem BUMP-Befehl können wir einfach feststellen, welche Sprites in eine Kollision verwickelt sind:

BUMP(A)

erschließt über die Typkennung A zwei Sorten von Zusammenstößen.

A=1 Sprite/Sprite-Kollision
A=2 Sprite/Text-Kollision

Der Befehl hat lediglich die Aufgabe, den Inhalt eines speziellen Kollisionsregisters auszulesen. Das allerdings macht er so gründlich, daß dieses Register hinterher gelöscht ist. Es empfiehlt sich daher, diesen Wert sogleich in eine Variable einzuspeichern:

V = BUMP(1)

Die Zahl, die auf diese Weise erhalten wird, muß allerdings erst entschlüsselt werden (siehe da, ein alter Bekannter aus dem C64-Modus), denn jedem Sprite ist ein Bit zugeordnet: Sprite 1 hängt mit Bit 0, Sprite 2 mit Bit 1 zusammen und so weiter bis zu Sprite 8, welches mit Bit 7 geht. Findet nun die Kollision statt, dann schalten die Bit der tangierten Sprites auf 1. Bild 2.3 zeigt Ihnen diese Zusammenhänge und einige Rechenbeispiele:

Bits:	7	6	5	4	3	2	1	0	
Bitwerte:	128	64	32	16	8	4	2	1	
Sprite:	8	7	6	5	4	3	2	1	
Beispiele:	0	0	0	0	0	0	1	1	= 3, also Sprites 1 und 2
	0	0	0	0	0	1	0	1	= 5, also Sprites 1 und 3
	0	0	0	0	1	0	0	1	= 9, also Sprites 1 und 4
	1	0	0	0	0	0	0	1	= 129, also Sprites 1 und 8
	1	1	0	0	0	0	0	0	= 192, also Sprites 7 und 8

Bild 2.3: Das Ergebnis der BUMP(A)-Abfrage

Ein Supercrash unter Beteiligung aller 8 Sprites würde dann also die Zahl 255 ergeben. Ein kleines Testprogramm 2 soll das Vorgehen dabei illustrieren:

Testprogramm 2

Ein Programm zur BUMP-Programm

```
10 REM ***** TESTPROGRAMM 2 BUMP-BEFEHL *****
20 BLOAD"SPRITES2",ONB0
30 SPRCOLOR3,6
40 MOVSPR7,100,100:MOVSPR8,150,200
50 SPRITE7,1,2,0,0,0,1:SPRITE8,1,1,1,1,1,1
60 V=BUMP(1):IF V>0 THEN GOSUB85
70 MOVSPR8,+1,+3
80 GOTO60
85 PRINT:PRINT"BUMP(1) ERGIBT DEN WERT"V
90 FORI=0TO7:A=2↑I:IF A AND V THEN PRINT"SPRITE "I+1"KOLLIDIERT
100 NEXTI
110 RETURN
```

Besonders die Zeilen 90 und 100 sind sicherlich für Sie interessant, weil man mit ihnen die Nummern der beteiligten Sprites feststellen kann.

Auch hier gilt, was schon bei COLLISION bemerkt wurde: Auch Zusammenstöße außerhalb des sichtbaren Bildschirmes werden registriert und das auch ohne 40-Zeichen-Bildschirm. Ebenso wie dort muß beachtet werden, daß ein Zusammenstoß einige Zeit in Anspruch nimmt, also die Reaktion darauf meistens mehrfach erfolgt.

Im Gegensatz allerdings zu COLLISION, das interruptgesteuert abläuft, muß die Abfrage des BUMP-Wertes mehrfach erfolgen. Der Befehl ist sowohl ohne COLLISION als auch damit einsetzbar. Im letzteren Fall dient er als Ergänzung im Unterprogramm zur Feststellung der Spritenummern.

2.1.1.8 RSPCOLOR

Damit sind Informationen zu den aktuellen Multicolor-Register-Inhalten zu bekommen:

RSPCOLOR(A)

liefert bei

A = 1	Farbcode in Multicolorregister 1 (dasjenige, welches der Bitkombination 01 zugeordnet ist);
A = 2	hier wird der Inhalt des anderen Multicolorregisters ausgegeben, das zu der Bitkombination 11 gehört.

2.1.1.9 RSPPOS

Sollte uns die aktuelle Position oder Geschwindigkeit eines Sprites interessieren, dann können wir das durch

RSPPOS(n,A)

erfahren. Dabei ist n die Spritenummer und A wieder eine Kennung mit folgender Zuordnung:

A = 0	aktuelle X-Koordinate
A = 1	Y-Koordinate
A = 2	gerade vorhandene Geschwindigkeit

Leider gibt es da eine kleine Unstimmigkeit gegenüber dem Befehl MOVSPR: Gleichgültig, welches Sprite-Koordinatensystem wir dort durch SCALE festgelegt haben, RSPPOS liefert immer die Werte des normalen Systems. RSPPOS(0) liegt daher immer zwischen 0 und 511, RSPPOS(1) immer zwischen 0 und 255. Sollten Sie bei normalem System mittels MOVSPR höhere Werte eingegeben haben, dann wird hier immer modulo(512) für die X- und modulo(256) für die Y-Koordinate ausgegeben.

Auch die Sache mit der aktuellen Geschwindigkeit hat einen Haken: Falls Sie nämlich den Sprite mal auf andere Weise als durch MOVSPR n,W#V bewegen, erhalten Sie den Wert 0 als Ausgabe von RSPPOS(2).

2.1.1.10 RSPRITE

Hier haben wir das Pendant zum SPRITE-Befehl vorliegen. Alle dort verwendeten Parameter können hier durch

RSPRITE(n,A)

abgefragt werden. n ist wieder die Spritenummer, A eine Kennung mit den Zuordnungen:

A = 0	Sprite ein- (=0) oder ausgeschaltet (=1)
A = 1	Farbcode des Sprites (im Multicolormodus Farbe der Bitkombination 10)
A = 2	Sprite vor (=0) oder hinter (=1) Text
A = 3	In X-Richtung gedehnt (=1) oder nicht (=0)
A = 4	In Y-Richtung gedehnt (=1) oder nicht (=0)
A = 5	Multicolormodus (=1) oder normales Hires-Sprite (=0)

2.1.1.11 JOY

Dieser Befehl hat zwar nicht unbedingt etwas mit den Sprites zu tun, wird aber häufig zum Steuern der Sprites eingesetzt und kommt daher an diese Stelle. Man kann damit auf einfache Weise den Zustand der Joystickports abfragen:

JOY(A)

fragt bei A = 1 den Port 1,

bei A = 2 den Port 2 ab.

Bevor wir uns die Bedeutung der Abfrageergebnisse ansehen, noch eine Warnung: Vielleicht lag es an meinem Computer, vielleicht ist es aber auch ein allgemeiner Fehler, daß die Verwendung von JOY(1) in Programmen häufig dazu führte, daß während der Abfrage plötzlich der Ablauf im Monitor mit einer Registeranzeige endete. Mit JOY(2) lief dagegen alles einwandfrei.

Die mittels JOY ausgelesenen Werte haben folgende Bedeutung:

0	Ruhestellung
1-8	Richtungen (siehe Bild 2.4)
128	Ruhestellung, Feuerknopf gedrückt
129-136	Richtungen mit gedrückten Feuerknopf (siehe Bild 2.4)

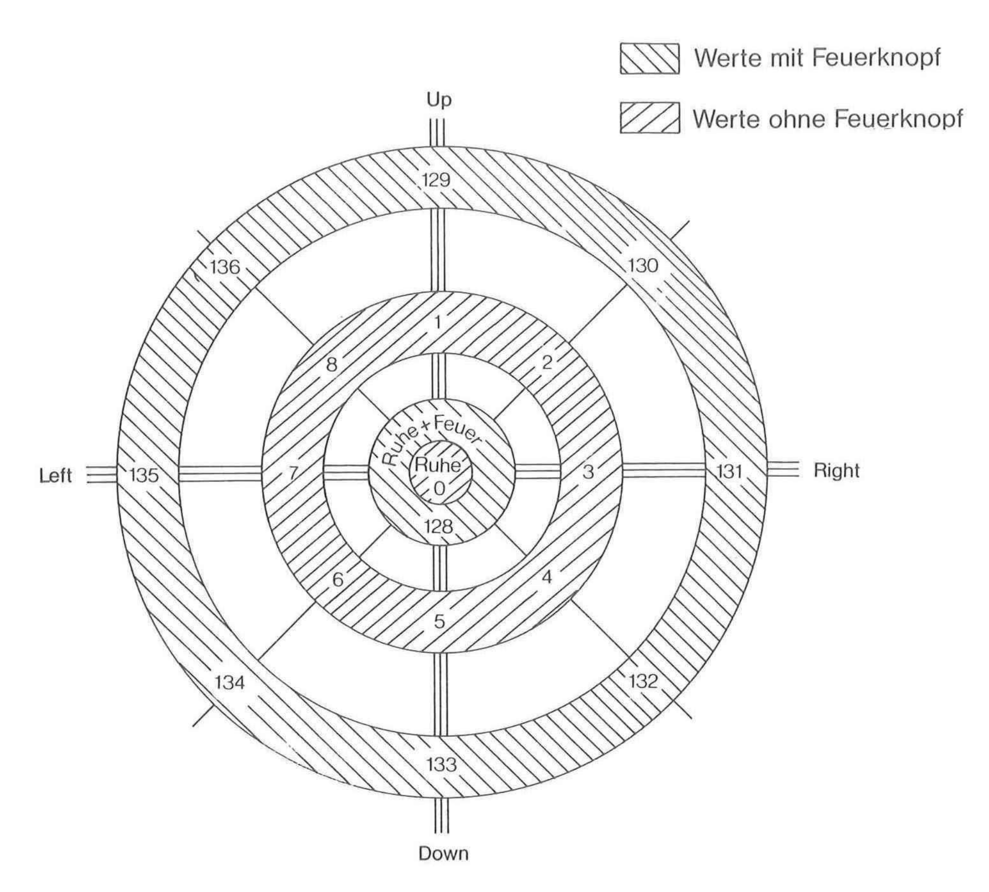


Bild 2.4: Schema zur JOY-Abfrage

Auf diese Weise können wir bequem beispielsweise Sprites steuern mit dem Joystick:

```
100 ON JOY(2) GOSUB 300,310,320,330,...
```

Im Testprogramm 3 ist eine mögliche Variante dazu gezeigt.

Testprogramm 3

Steuern mittels JOY

```
10 REM ***** TESTPROGRAMM 3  JOY-BEFEHL *****
20 BLOAD"SPRITES2",ONB0
30 SPRCOLOR3,6
40 MOVSPR1,100,100:MOVSPR8,200,200
50 SPRITE1,1,2,0,0,0,1:SPRITE8,1,1,1,1,1,1
60 COLLISION1,110
70 X=200:Y=200
80 ONJOY(2)GOSUB190,200,210,220,230,240,250,260
90 MOVSPR8,X,Y
100 GOTO80
110 COLLISION1
120 PRINT"KOLLISION ERFOLGT"
130 COLLISION1,150
140 RETURN
150 COLLISION1
160 PRINT"NOCH EINE KOLLISION"
170 COLLISION1,150
180 RETURN
190 X=X:Y=Y-1:RETURN
200 X=X+1:Y=Y-1:RETURN
210 X=X+1:Y=Y:RETURN
220 X=X+1:Y=Y+1:RETURN
230 X=X:Y=Y+1:RETURN
240 X=X-1:Y=Y+1:RETURN
250 X=X-1:Y=Y:RETURN
260 X=X-1:Y=Y-1:RETURN
```

Damit haben wir alle im BASIC 7.0 enthaltenen Sprite-Befehle behandelt. Allerdings sind noch einige Fragen offen geblieben, die uns nun weiter beschäftigen werden.

2.1.2 Wie kann man Sprites der Nachwelt erhalten?

2.1.2.1 Sprites im Speicher

a) Sprite-Daten-Speicher

Schon beim SPRDEF-Befehl haben wir diesen Speicherbereich zwischen 3584 und 4095 ($0E00 bis 0FFF) erwähnt, in dem die Spritedaten landen, wenn ein Sprite durch Shift-Return gespeichert wird. Pro MOB befinden sich dort - in der Reihenfolge der Spritenummern - je 63 Byte und eine Endmarkierung (das ist ein 0-Byte). Verfügbar bleiben diese Daten, bis sie überschrieben werden oder der Computer abgeschaltet wird.

b) Sprite-Strings

Noch flüchtiger ist die Existenz der Daten im String, der durch SPRSAV gebildet wird. Schon ein CLR macht der Kunst den Garaus. In jeweils 67 Byte (die letzten 4 enthalten immer die Werte 23, 0, 20, 0) finden sich dort die Bitmuster. Sollten Sie einmal versuchen, so einen String (z.B. A$) durch PRINT A$ auf dem Bildschirm zu zeigen, dann müssen Sie sich auf allerhand Überraschungen gefaßt machen: Alle Bytewerte von 0 bis 255 können auftreten und wirken so, als wären sie mittels CHR$(Wert) zur Ausgabe aufgerufen worden. Wenn also der Wert 147 zufällig dabei sein sollte, dann wird der Bildschirm gelöscht, etc. Jedes RUN führt übrigens ebenfalls ein CLR aus. Auch das ist daher nicht die richtige Methode, die mühselig konstruierten Sprites etwas länger am Leben zu erhalten.

Sowohl im Speicher als auch im String haben die Sprite-Daten genau dasselbe Format (bis auf die letzten Byte, die im einen Fall aus einer 0, im anderen aus den Zahlen 23,0,20,0 bestehen). Das können Sie selbst nachprüfen am beigefügten Programm "VERGLEICH SPRDT".

Programm VERGLEICH SPRDT

Programm zum Vergleichen von Spritemustern im Speicher und im String

```
10 REM ***** VERGLEICH SPRITEDATEN *****
20 REM IN SPRITESPEICHER UND IM STRING
30 DIMA$(7)
40 BLOAD"SPRITES1",ONB0
50 FORJ=0TO7
60 :PRINTCHR$(147)CHR$(18)J+1".SPRITE"CHR$(146):PRINT:SPRSAVJ+1,A$(J)
70 :BANK0:PRINT"DATENSPEICHER:"
80 :FORI=0TO63
90 ::PRINT(PEEK(3584+64*J+I));
100 :NEXTI:PRINT:PRINT"STRING:"
110 :FORK=1TOLEN(A$(J))
120 ::PRINTASC(MID$(A$(J),K,1));
130 :NEXTK:PRINT:PRINT"BITTE TASTE DRUECKEN"
140 :GETKEYB$
150 :PRINT
160 NEXTJ
```

Nach dem Start werden zunächst die Spritedaten von der Diskette geladen und dann in Strings eingelesen. Anschließend erscheinen auf dem Bildschirm Sprite für Sprite die Bytewerte sowohl aus dem Speicher als auch aus dem String.

2.1.2.2 Sprites auf Diskette und Kassette

a) BSAVE

Eine elegante Methode zur Verewigung unserer Sprites ist das Abspeichern des Spritespeichers mittels des BSAVE-Befehls. Man schreibt dazu:

BSAVE"Name",ON B0,P3584 TO P4095

Damit sind dann alle 8 Sprites erfaßt. Leider arbeitet dieser Befehl nicht mit der Kassettenstation, denn es sind nur Gerätenummern von 4 bis 15 zulässig. Das Wiedereinladen geschieht dann (wie in den hier vorgestellten Beispielprogrammen) mittels

BLOAD"Name",ON B0

b) Vom Monitor aus

Damit ist es nun auch Benutzern der Datasette möglich, den Spritespeicher zu sichern auf Kassette. Der Monitor verfügt über ein Kommando S zum Speichern beliebiger Speicherbereiche. Mittels MONITOR oder der Funktionstaste F8 schalten Sie den Monitor an, dann verwenden Sie:

S"Name",01,+3584,+4095 Kassette oder
S"Name",08,+3584,+4995 Diskette.

Mit Hilfe des L-Kommandos im Monitor kann solch ein File dann problemlos wieder geladen werden:

L"Name",01(oder 08),+3584

c) Als sequentiellen File

Hat man die Sprites in Strings abgelegt (das dürfen dann auch mehr als 8 sein), dann kann man sich der üblichen Techniken zur Speicherung in sequentiellen Dateien bedienen. Allerdings kann es manchmal dabei Schwierigkeiten mit bestimmten Byte-Inhalten geben. So ist es durchaus möglich, daß ein Byte zufällig den Inhalt 13 (also einem RETURN entsprechend) hat, was zu Störungen beim Wiedereinlesen führen kann. In solchen Fällen könnte man die einzelnen Byte (wie im Programm VERGLEICH SPRDT geschehen) in die ASCII-Zahlen wandeln und in dieser Form als SEQ-File speichern. Diese Möglichkeit werde ich nicht weiter beschreiben, weil mir die Speicherung per Monitor oder BSAVE schneller und effektiver erscheint.

2.1.2.3. Für ein Listing

Viele Programme werden nicht in Form von Disketten- oder Kassettenfiles, sondern einfach auf dem Papier weitergegeben. Dieses Problem lösen bessere Sprite-Editoren durch eine Funktion, die alle Werte in DATA-Zeilen an ein Programmende anhängt. In SPRDEF existiert diese Möglichkeit leider nicht, was uns dazu zwingt, selbst die Initiative zu übernehmen.

a) Aus dem Speicher in DATAs

Dazu habe ich Ihnen ein kleines Programm gestrickt (SPRITEDATAS),

das Sie mit der MERGE-Funktion (an anderer Stelle im Buch präsentiert) an Ihr eigenes fertiges Spriteprogramm anhängen können. Noch eine Warnung: Wenn Sie SPRITDATAS abgetippt haben, speichern Sie es unbedingt vor einem RUN ab, denn es verabschiedet sich am Ende des Programmlaufes aus Ihrem BASIC-Speicher. Nun also zur Verwendung des Programms. Man startet es durch RUN63000. Das Auslesen des Sprite-Datenspeichers geschieht im FAST-Modus (der 40-Zeichen-Bildschirm verabschiedet sich vorübergehend), danach erscheinen jeweils 2 Programmzeilen (mit Zeilennummern ab 63020) und darunter ein GOTO 63009. Außerdem meldet sich ein BREAK und READY. Mittels der HOME-Taste und 3 aufeinanderfolgenden RETURNs übernehmen Sie die Zeilen ins Programm und erzeugen den nächsten Bildschirm, wo das Spielchen dann ebenso weitergeht. Insgesamt machen wir das 8 mal (für 8 Sprites). Zu guter Letzt wird in Zeile 63040 noch eine Einleseschleife übernommen, und es meldet sich der Befehl DELETE 63000-63011. Haben Sie das auch mit HOME und 2 RETURNs übernommen, dann ist der DATA-Generator gelöscht, und an Ihr Programm wurde die gesamte DATA-Sequenz angehängt. Als letzte Zeile finden Sie beim Listen noch die Einleseschleife, zu der Sie nun nur noch an passender Stelle Ihres Programms mittels GOSUB 63040 springen müssen.

Programm SPRITEDATAS

Erzeugen von DATA-Zeilen aus Spritedaten

```
62999 REM *** DATAS AUS SPRITESPEICHER ***
63000 FAST:BANK0:FORJ=0TO7
63001 A$(J)="":FORI=0TO62:A$=STR$(PEEK(3584+64*J+I))
63002 A$=RIGHT$(A$,LEN(A$)-1):A$="000"+A$:A$=RIGHT$(A$,3)
63003 A$(J)=A$(J)+A$+",":NEXTI
63004 A$(J)=A$(J)+"000"
63005 NEXTJ:SLOW
63006 FORJ=0TO7:PRINTCHR$(147);
63007 PRINT63020+2*J"DATA"LEFT$(A$(J),127)
63008 PRINT63021+2*J"DATA"RIGHT$(A$(J),127):PRINT"GOTO63009":STOP
63009 NEXTJ:PRINTCHR$(147);
63010 PRINT"63040 FORI=3584TO4095:READA:POKEI,A:NEXTI:RETURN"
63011 PRINT"DELETE 63000-63011":STOP

READY.
```

Es gibt sicherlich elegantere Möglichkeiten, DATA-Zeilen zu generieren. So könnten wir auch den Tastaturpuffer verwenden, wie es an anderer Stelle dieses Buches geschildert wird. Das überlasse ich Ihrer Schöpferkraft: Mir lag mehr an der Überschaubarkeit des Programms.

b) Aus den Strings in DATAs

Auch das Übertragen der Stringinhalte in DATA-Zeilen ist eine Lösungsmöglichkeit unseres Problems. Sie ist sogar sehr verlockend, weil wir damit auf einen Streich mehr als 8 Sprites ins Listing schreiben könnten. Die Programmierung wird aber etwas komplizierter, weil hier nicht mehr der ganze Inhalt des Sprite-Strings in einen Zeilen-String gelesen werden kann. Es gibt eine Reihe von Strategien, das Problem zu lösen - mehr oder weniger einfache, was teilweise mit dem Auslesen der DATAs zusammenhängt -, auch hier sind Sie gefordert!

2.1.3 Koppeln von Sprites

Sollten Sie ein komplexeres grafisches Gebilde erzeugen wollen als mit einem Sprite möglich ist, dann können Sie auch mehrere Sprites koppeln. Am Beispiel von 4 Sprites zeigt Ihnen Bild 2.5 die dazu nötigen Koordinatenwerte:

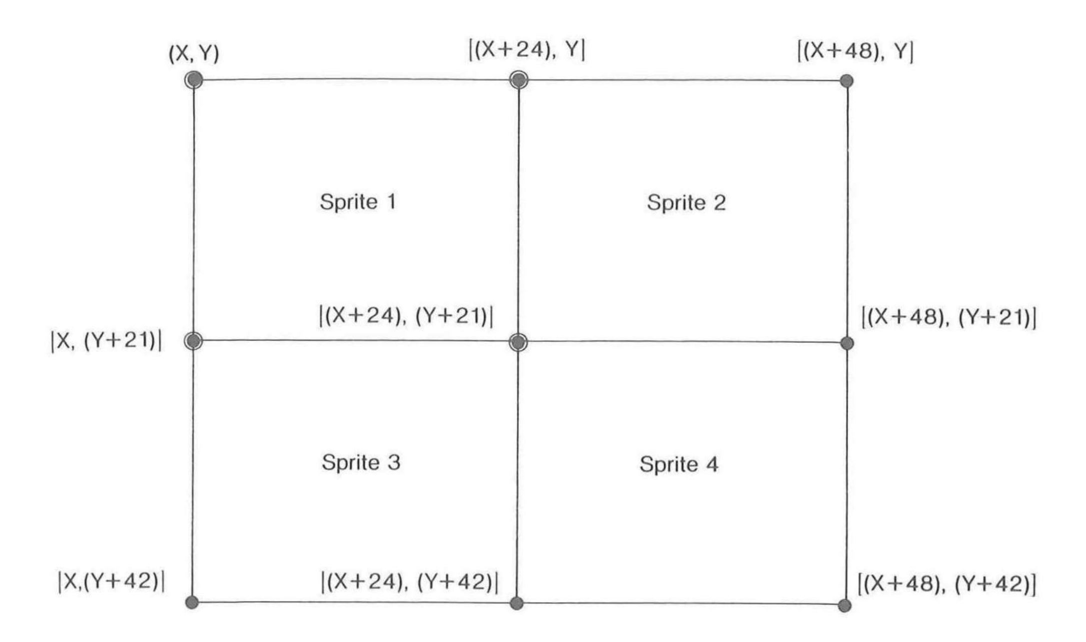


Bild 2.5: Koppeln von 4 Sprites

Das Prinzip ist simpel: Weil jedes Sprite in X-Richtung 24 Bit breit ist, ergibt sich für die X-Koordinate des Nachbarsprites bei nahtloser Verbindung ein um 24 größerer Wert. 21 Bit mißt ein MOB in der Vertikalen, wodurch sich Y+21 für ein unterhalb des ersten gelegenes Sprite berechnet als Y-Koordinate. Das folgende Programm "VIER SPRITES" demonstriert die Verhältnisse.

Programm VIER SPRITES

Demonstration zum Koppeln von Sprites

```
10 REM ***** VIER SPRITES *****
20 GOTO110
30 Y=Y-3:RETURN
40 X=X+3:Y=Y-3:RETURN
50 X=X+3:RETURN
60 X=X+3:Y=Y+3:RETURN
70 Y=Y+3:RETURN
80 X=X-3:Y=Y+3:RETURN
90 X=X-3:RETURN
100 X=X-3:Y=Y-3:RETURN
110 BLOAD"SPRITES2",ONB0
120 PRINTCHR$(147):COLOR0,1:COLOR4,1:COLOR5,8
130 X=50:Y=60
140 MOVSPR3,X,Y:MOVSPR4,X+125,Y:MOVSPR5,X,Y+122:MOVSPR6,X+125,Y+122
150 SPRITE3,1,8,1:SPRITE4,1,3,1:SPRITE5,1,8,1:SPRITE6,1,8,1
160 DO UNTIL A=101
170 :MOVSPR4,-1,Y:MOVSPR5,X,-1:MOVSPR6,-1,-1
180 :A=A+1
190 LOOP
200 BANK15:SYS65520,10,20:A$="JOYSTICK IN PORT2"
210 DO UNTIL JOY(2)=128
220 :COLOR5,3
230 :GOSUB330
240 :COLOR5,1
250 :GOSUB330
260 LOOP:X=RSPPOS(3,0):Y=RSPPOS(3,1)
270 COLOR5,8:PRINTCHR$(147):FORI=1TO50:NEXT
280 DO
290 :ONJOY(2)GOSUB30,40,50,60,70,80,90,100
300 :MOVSPR3,X,Y:MOVSPR4,X+24,Y:MOVSPR5,X,Y+21:MOVSPR6,X+24,Y+21
310 LOOP UNTIL JOY(2)=128
320 END
330 SYS65520,10,20:PRINTA$:FORK=1TO50:NEXTK:RETURN
```

Zunächst wächst ein grafisches Objekt aus vier Sprites zusammen. Dieses kann dann mit dem Joystick in Port 2 (nach Druck auf den Feuerknopf) über den Bildschirm gesteuert werden (Zeile 300). Ein weiterer Druck auf den Feuerknopf beendet das Programm. Wenn Sie bei bestimmten Richtungen genau hinsehen, werden Sie eine leichte Unstimmigkeit in der Bewegung feststellen. Bei noch größeren Objekten scheint BASIC für diese Steuerung doch etwas zu langsam zu werden, und man muß auf Maschinensprache umsteigen, um eine flüssige Bewegung zu erreichen.

2.1.4 Mehr als 8 Sprites

Hier soll nicht die Rede sein von trickreichen Anwendungen der Unterbrechungsprogrammierung per Assembler, die es tatsächlich zuläßt, mehr als 8 Sprites **gleichzeitig** auf dem Bildschirm zu zeigen. Vielmehr geht es uns um die schnelle, nacheinander ausgeführte Abbildung. Sehen Sie sich dazu zuerst einmal das Programm "SPRITE-TRICK" an.

Programm SPRITE-TRICK

Ein kleiner Trickfilm

```
10 REM ***** EIN SPRITE-TRICK *****
20 BLOAD"SPRITES1",ON B0
30 PRINTCHR$(147)
40 FOR I=1 TO 8:MOVSPRI,200,100:NEXTI
50 FOR I=1 TO 10:FOR J=1 TO 8
60 IF J=1 THEN SPRITE8,0:ELSE SPRITEJ-1,0
70 SPRITEJ,1,I+1,1,1,1,0
80 FOR K=1 TO 25:NEXTK:NEXTJ
90 NEXTI
100 SPRITE8,0:SPRITE1,0
```

Hier liegen 8 Spritemuster vor, die nacheinander - mit programmierter Verzögerung, weil's sonst zu schnell ginge - auf derselben Bildschirmstelle gezeigt werden. Es ergibt sich ein kleiner Trickfilm.

Nun sind 8 Teilbilder ein etwas ärmlicher Bewegungseffekt. Um wirklich längere Passagen zu zeigen, müßten wir anders verfahren. Das soll im Programm "24 SPRITES" demonstriert werden.

Programm 24 SPRITES

Ein Trickfilm mit 24 Sprites

```
10 REM ***** 24 SPRITES *****
20 I=0:J=1:K=2:L=3:M=4:N=5:O=6:P=7:B$="":DIMA$(24)
30 BLOAD"SCENE1"
40 FOR I=1 TO 8:SPRSAVI,A$(I):NEXTI
50 BLOAD"SCENE2"
60 FOR I=1 TO 8:SPRSAVI,A$(I+8):NEXTI
70 BLOAD"SCENE3"
80 FOR I=1 TO 8:SPRSAVI,A$(I+16):NEXTI
90 PRINTCHR$(147)
100 SPRCOLOR3,6:COLOR0,1:COLOR4,1:COLOR5,8
110 MOVSPR1,62,86:MOVSPR2,111,86:MOVSPR3,160,86:MOVSPR4,209,86:MOVSPR5,62,129
120 MOVSPR6,111,129:MOVSPR7,160,129:MOVSPR8,209,129
130 SPRITE1,1,8,1,1,1,1:SPRITE2,1,14,1,1,1,1:SPRITE3,1,2,1,1,1,1
140 SPRITE4,1,11,1,1,1,1:SPRITE5,1,13,1,1,1,1:SPRITE6,1,5,1,1,1,1
150 SPRITE7,1,1,1,1,1,1:SPRITE8,1,16,1,1,1,1
160 I=0
170 DO
180 :I=I+1:IF I>24 THEN I=1:SPRSAVA$(I),1:ELSE SPRSAVA$(I),1
190 :J=J+1:IF J>24 THEN J=1:SPRSAVA$(J),2:ELSE SPRSAVA$(J),2
200 :K=K+1:IF K>24 THEN K=1:SPRSAVA$(K),3:ELSE SPRSAVA$(K),3
210 :L=L+1:IF L>24 THEN L=1:SPRSAVA$(L),4:ELSE SPRSAVA$(L),4
220 :M=M+1:IF M>24 THEN M=1:SPRSAVA$(M),5:ELSE SPRSAVA$(M),5
230 :N=N+1:IF N>24 THEN N=1:SPRSAVA$(N),6:ELSE SPRSAVA$(N),6
240 :O=O+1:IF O>24 THEN O=1:SPRSAVA$(O),7:ELSE SPRSAVA$(O),7
250 :P=P+1:IF P>24 THEN P=1:SPRSAVA$(P),8:ELSE SPRSAVA$(P),8
260 :GETB$:IF B$="←" THEN EXIT
270 LOOP
280 END
```

Das Geheimnis - Sie haben es sicherlich schon längst vermutet - liegt im SPRSAV-Befehl begründet. 24 Spritemuster wurden hier in ein Stringarray eingelesen (Zeilen 20 bis 80). Während die Sprites auf dem Bildschirm aktiv sind (die Aktivierung erfolgt schon in den Zeilen 130 bis 150), wechseln wir durch die Zuordnung anderer Spritemuster aus den Strings ihre Erscheinungsformen. Das geschieht in der DO...LOOP-Schleife ab Zeile 170. Hier geschieht das - wegen des hübschen Mustereffektes - auf zyklische Art und Weise (durch Drücken von _ können Sie das Programm beenden), man kann sich aber ohne weiteres vorstellen, wie ein oder mehrere große Arrays gebildet und dann fortlaufend deren Spritemuster verwendet werden können. Gleichzeitig sind die sich ändernden Sprites noch beweglich. Sie sehen, daß kaum Grenzen gesetzt scheinen, allerhand Interessantes auf dem Bildschirm ablaufen zu lassen.

2.1.5 Noch ein paar Sprite-Besonderheiten

2.1.5.1 Sprites und Splitscreens

Beim Commodore 128 haben wir ja in den Grafik-Modi 2 und 4 die Möglichkeit, sowohl hochauflösende (oder aber Multicolor-) Grafik als auch Text auf einem Bildschirm gleichzeitig darzustellen. Damit ist nicht etwa der CHAR-Befehl gemeint, sondern durch eine spezielle Technik, die den Rasterzeileninterrupt ausnutzt, wird der Bildschirm tatsächlich in zwei verschiedenen Modi betrieben.

Wie verhalten sich nun die Sprites auf solch einem Bildschirm? Wie es Kobolden ansteht, scheren sie sich überhaupt nicht darum! Probieren Sie mal das Programm VIER SPRITES mit einer kleinen Änderung aus: In die Zeile 130 fügen Sie noch ein GRAPHIC 2,0,12 (oder für den Multicolormodus GRAPHIC 4,0,12) und starten dann. Ohne Reaktion von seiten der Sprites können Sie diese über die in der Bildmitte liegende Modusgrenze hinwegsteuern. Allerdings reagiert der Bildschirm mit einem penetranten Flattern der Grenzlinie, wenn das Sprite genau darauf liegt.

Falls Sie dasselbe mit dem Grafik-Modus 4 (also GRAPHIC 4,0,12) probieren, würde es mich interessieren, ob auch Ihr Computer nach einiger Zeit mit irgendeiner Fehlermeldung aussteigt. Bei meinem Modell verhielt er sich so. Wenn ich mir dann die als fehlerhaft gemeldete Zeile listen ließ, befanden sich allerlei merkwürdige - anscheinend zufällig erzeugte - Fehler darin. Zuvor aber war die Zeile fehlerfrei gewesen! Man sollte offensichtlich diese Möglichkeit mit etwas Vorsicht gebrauchen.

2.1.5.2 Sprites und Windows

Ebensowenig wie sich Sprites um den Splitscreen kümmern, scheren sie sich um Windows. Probieren Sie es mal aus, indem Sie in Zeile 130 des Programms VIER SPRITES noch ein Window definieren (zum Beispiel mit WINDOW 5,5,20,10). Unbeirrt von allen Einschränkungen, denen unser Text unterworfen ist (hier die Aufforderung zum Benutzen des Port 2), läßt sich unser Spritegebilde kreuz und quer verschieben.

Die Sprites an sich sind damit zunächst einmal durchleuchtet. Legen wir sie also für eine Weile beiseite und wenden wir uns den anderen Grafikobjekten des Commodore 128 zu, den Shapes.

2.2 SHAPES

2.2.1 Was sind Shapes?

Wie so vieles aus der Computerkultur stammt auch dieses Wort aus dem Angelsächsischen: Shape heißt ins Deutsche übersetzt soviel wie Form, Gestalt, Umriß. Hier bezeichnet Shape einen genau definierten Bildschirmausschnitt, der gespeichert und wiederverwendet werden kann. Wie das mit dem Commodore 128 geschieht, soll nun erklärt werden.

2.2.2 Die Shape-Befehle

2.2.2.1 SSHAPE

Dies ist der Befehl, mit dem ein Bildschirmbereich in einen String eingelesen werden kann. Seine Syntax ist

SSHAPE A$,X1,Y1,X2,Y2

A$ ist hier dann eine beliebige Stringvariable, auch ein Array kann Verwendung finden. X1 bis Y2 bezeichnet die Eckkoordinaten des Rechtecks, dessen Inhalt als Shape definiert werden soll. Bild 2.6 soll das etwas verdeutlichen:

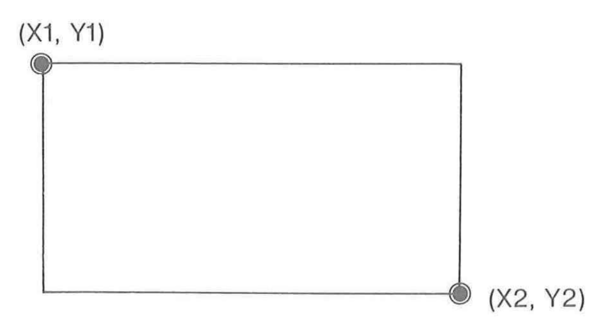


Bild 2.6: Zur Syntax des SSHAPE-Befehls

(X1,Y1) gehören zum linken oberen und (X2,Y2) zum rechten unteren Eckpunkt. Es ist auch möglich, X2 und Y2 wegzulassen. In diesem Fall werden durch den SSHAPE-Befehl einfach die aktuellen Koordinaten des Grafik-Cursors eingesetzt.

Falls es noch unklar sein sollte: Im Gegensatz zu Sprites, die jedem Bildschirm-Modus trotzen (demnach sowohl als Hires- oder Multicolorsprites im Text- als auch im Hochauflösungs- und im Multicolormodus auftauchen können), sind Shapes fest mit dem aktuellen Modus verbunden. Ein Shape ist ein Grafikobjekt, kann daher auch nur in den Grafikmodi auftreten und definiert werden. Also in denen, die durch GRAPHICn erzeugt werden, wobei n von 1 bis 4 geht. Außerdem ist ein Multicolorshape auch nur im Multicolormodus ein solches. Schaltet man beispielsweise mittels GRAPHIC1 um in den normalen Hochauflösungsmodus, wird auch unser Shape nur als hochaufgelöstes sichtbar.

Wie wir uns aus all dem schon fast denken können, ist das Koordinatensystem, auf das sich die Angaben X1 bis Y2 beziehen, das im jeweiligen Modus gültige. Im Normalfall also im Hires-Modus X von 0 bis 319 und Y von 0 bis 199. Im Multicolormodus geht X nur von 0 bis 159.

A$ ist eine Stringvariable, Strings dürfen beim Commodore 128 maximal 255 Byte lang sein. Ein durch SSHAPE in einen String zu schreibendes Gebiet darf also eine gewisse Größe nicht überschreiten. Bezeichnen wir als

DX = X2 - X1 und als
DY = Y2 - Y1

dann gilt für die Länge eines Shapes im Hochauflösungsmodus die Formel:

Lh=INT((ABS(DX)+1)/8+.99)*(ABS(DY)+1)+4

Im Multicolormodus gilt statt dessen die Gleichung:

Lm=INT((ABS(DX)+1)/4+.99)*(ABS(DY)+1)+4

Die Addition von 4 am Ende der Formeln ergibt sich durch vier Byte, die den Schluß des Shape-Strings bilden beim Eintrag des Strings in das Variablenfeld gebraucht werden.

Natürlich können Sie jedesmal, wenn Sie ein Shape erstellen, von dem Sie argwöhnen, es könne zu groß sein für den String, unsere Formeln anwenden. Eine Hilfe soll Ihnen das folgende Programm SSHAPE-TABELLE geben.

Programm SSHAPE-TAB

Eine Hilfe für Shapes: Eine Tabelle wird gedruckt

```
10 REM ***** SSHAPE TABELLE *****
20 FAST:OPEN1,4:PRINT#1,"Y-DIFF.","X-DIFF.","STRINGLAENGE"
30 DEFFNH(X)=INT((ABS(X)+1)/8+.99)*(ABS(Y)+1)+4
40 Y=0
50 DO UNTIL Y=200
60 :Y=Y+1:X=0
70 :DO UNTIL X=320
80 ::X=X+1
90 ::Z=FNH(X):IFZ>255THENPRINT#1,Y1,X1,Z1:EXIT
100 ::X1=X:Y1=Y:Z1=Z
110 :LOOP
120 LOOP
130 CLOSE1:SLOW:END
```

In der vorliegenden Form druckt es eine Liste aus, in deren linker Spalte Werte für eine Y-Differenz stehen. Rechts daneben sind die maximal zulässigen DX-Werte angegeben sowie die sich ergebende Stringlänge. Auf diese Weise können Sie - ausgehend von DY - mit einem Blick erkennen, wie groß DX sein darf. Es empfiehlt sich, noch 3 weitere Tabellen auszudrucken. Da ist zunächst einmal interessant, wie der Maximalwert von DY bei gegebenen DX aussieht. Zu diesem Zweck ist lediglich die Reihenfolge des Ausdruckes und der Rechenoperationen zu ändern: In Zeile 20 kommt dann X-DIFF vor Y-DIFF, Zeile 40 enthält X=0, in Zeile 50 schreiben wir DO UNTIL X=320, in der nächsten Zeile 60 steht: X=X+1:Y=0. Zeile 70 wird zu DO UNTIL Y=200 und Zeile 80 zu Y=Y+1. Schließlich wird auch in Zeile 90 noch die Reihenfolge des Ausdruckes umgestellt zu PRINT#1, X1,Y1,Z.

Diese beiden Tabellen beziehen sich auf Shapes im Hochauflösungsmodus. Zwei weitere Tabellen für den Multicolormodus bekommen Sie, wenn Sie in Zeile 30 die vorhin angegebene Berechnungsfunktion zu diesem Modus einfügen. Nur die Zahl 8 muß dazu in eine 4 verändert werden.

Sollte dann, trotz all dieser Berechnungsmöglichkeiten - man sieht ja nicht jedesmal in die Tabellen - der zu speichernde Shape größer werden als 255 Byte, dann meldet unser Computer einen STRING TOO LONG ERROR.

Einen Aspekt zum SSHAPE-Befehl müssen wir noch besprechen, weil es da einige Verwirrung geben kann: Wie reagiert SSHAPE auf ein durch SCALE verändertes Bildschirmsystem? Die Koordinatenangabe bei SSHAPE muß dann ebenfalls in den aktuell gültigen Werten geschehen. Man muß sich also nicht um die Organisation des Koordinatensystems kümmern, denn durch SSHAPE wird automatisch die Umrechnung auf den normalen Bildschirm vorgenommen. Lediglich bei der Berechnung der Stringlänge müssen einige Korrekturen eingefügt werden.

Nehmen wir an, wir hätten durch SCALE 1,XM,YM ein System definiert, dann liegen (im Hochauflösungsmodus) die Vergrößerungsfaktoren

FX = XM/320
FY = YM/200

vor, und die Beziehungen für die Stringlänge lauten dann:

Hochauflösungsmodus

```
Lh=INT((ABS(DX*320/XM)+1)/8+.99)* (ABS(DY*200/YM)+1)+4
```

Multicolormodus

```
Lm=INT((ABS(DX*160/XM)+1)/4+.99)* (ABS(DY*200/YM)+1)+4
```

DX und DY sind dabei im aktuellen Koordinatensystem geltende Werte. Begnügen wir uns nun mit dem SSHAPE-Befehl (es gäbe noch einiges zu untersuchen), und wenden wir uns dem Gegenstück, dem GSHAPE-Befehl zu.

2.2.2.2 GSHAPE

Ein durch SSHAPE im String gespeichertes Bild kann durch diesen Befehl nun wieder auf dem Bildschirm dargestellt werden:

GSHAPE A$,X,Y,M

A$ ist der uns nun schon bekannte Speicherstring, der durch SSHAPE definiert wurde. Es kann jede Stringvariable oder ein String-Array-Element verwendet werden.

X,Y sind die Koordinaten im aktuellen System, an die die linke obere Ecke unseres Shapes gelegt werden soll.

Interessant ist die Angabe von M. Hier dreht es sich um einen Darstellungsmodus, von dem es fünf Möglichkeiten gibt:

M = 0	Unser Shape wird genauso abgebildet, wie es definiert wurde. Schon an dieser Stelle auf dem Bildschirm vorhandene Objekte werden überdeckt. Dasselbe ergibt sich, wenn wir diesen Parameter einfach weglassen.
M = 1	Das Shape wird invertiert abgebildet. Auch hier überdeckt das Shape vorhandene Objekte.
M = 2	OR-Modus: Bildpunkte werden sichtbar, wenn sie zum Shape oder zu vorhandenen Objekten gehören. Beide Bilder überlagern sich so.
M = 3	AND-Modus: Bildpunkte werden nur dann gesetzt, wenn sie sowohl zum Shape als auch zum Bildschirmobjekt gehören.
M = 4	EOR- (oder XOR-) Modus: Bildpunkte werden nur dann gesetzt, wenn das entsprechende Bit für Shape und Bildschirmobjekt ungleich ist.

Die Modi 2, 3, 4 sind etwas schwer vorstellbar. Zur Verdeutlichung finden Sie hier ein kleines Demonstrationsprogramm namens GSHAPE-MODI:

Programm GSHAPE-MODI

Die fünf SHAPE-Darstellungsformen

```
10 REM *** DIE GSHAPE-MODI ***
20 REM ERSTELLEN EINES SHAPE
30 COLOR0,1:COLOR1,6:COLOR4,1:GRAPHIC1,1
40 CIRCLE1,10,10,10,10:BOX1,0,0,20,20:PAINT1,1,1:PAINT1,19,1:PAINT1,1,19
50 PAINT1,19,19:CIRCLE1,10,10,7,7:DRAW1,10,0TO10,20:DRAW1,0,10TO20,10
60 SSHAPEA$,0,0,20,20
70 SCNCLR1
80 REM BILDSCHIRMHINTERGRUND
90 DX=5:DY=5:X=0:Y=0
100 DO WHILE X<511
110 :X=X+DX:Y=Y+DY
120 :DRAW1,0,YTOX,0
130 LOOP
140 REM ABBILDEN DES SHAPE
150 FORI=0TO4
160 :GSHAPEA$,10+55*I,100,I
170 NEXTI
180 REM KOMMENTAR
190 COLOR1,3
200 FORI=0TO4
210 :B$=RIGHT$(STR$(I),1)
220 :CHAR1,2+7*I,18,B$
230 NEXTI
240 CHAR1,1,4,"DIE GSHAPE-MODI UND IHRE WIRKUNGEN:",1
250 CHAR1,1,23,"BITTE EINE TASTE"
260 GETKEYA$
270 GRAPHIC0,1
280 END

READY.
```

Hier wird zuerst ein Shape definiert, dann der gesamte Bildschirm quergestreift und schließlich das soeben erstellte Shape in allen 5 Modi darauf dargestellt. Die Wirkung ist auf diese Weise ganz gut zu erkennen, besonders bei den Modi 2, 3 und 4.

Der Modus 4 ist aufgrund der EOR-Behandlung interessant: Wird nämlich auf dieselbe Stelle noch einmal das Shape im Modus 4 gezeichnet, dann verschwindet es ganz, und der Bildschirmhintergrund ist wieder vorhanden. Will man Shapes bewegen, ohne Bildschirmobjekte zu zerstören, dann ist diese Darstellungsweise ganz gut dazu geeignet.

Die Frage, wie viele Shapes man definieren und auch darstellen könne, erübrigt sich fast: beliebig viele, solange der Speicherplatz reicht für die Strings. Und davon haben wir eine ganze Menge! Man könnte sich also eine ganze Bibliothek von Hochauflösungs- oder Multicolorobjekten zulegen und diese dann bei Bedarf abrufen. Sollte für ein Objekt der String zu kurz

sein, können - ebenso wie wir es bei den Sprites schon praktiziert haben - mehrere Shapes gekoppelt werden. Sie erkennen schon: Auch hier sind die Möglichkeiten sehr breit.

2.2.3 Zusammenspiel von SSHAPE und GSHAPE

Eine Frage haben wir noch offen gelassen, die eine recht verlockende Konsequenz in sich birgt: Wenn man mittels der SCALE-Anweisung die Systeme bei SSHAPE (also beim Aufbau des Shapes) und bei GSHAPE (bei der Abbildung) unterschiedlich wählt, kann man dann Shapes verkleinern oder vergrößern?

Die Antwort ist leider nein. Bei GSHAPE werden nur die Koordinaten eines (des linken oberen) Eckpunktes angegeben. Lediglich auf den Darstellungsort hat somit ein unterschiedliches Koordinatensystem eine Wirkung, nicht aber auf die Shape-Größe.

2.2.4 Bewegen von Shapes

Im Gegensatz zu den Sprites ist das Bewegen von Shapes eine langweilige Angelegenheit. Jedem Shape-Aufbau durch GSHAPE kann man ganz geruhsam zusehen. Zwar ist es durch Sequenzen wie

```
FOR I = 0 TO 200
GSHAPE A$,I,I,4
GSHAPE A$,I,I,4
NEXT I
```

möglich, Shapes ohne Schaden für den Bildschirminhalt über den Sichtbereich ziehen zu lassen. Das Ganze ähnelt aber bei weitem nicht der Sprite-Bewegung. Aus diesem Grund ist es anzuraten, bei der Bewegung von Bildschirmobjekten - wann immer möglich - Sprites zu wählen.

2.2.5 Shapes der Nachwelt erhalten

Wir stehen bei den Shapes demselben Problem gegenüber wie bei den Sprites. Auf welche Weise kann ein einmal erstelltes Shape für weiteren Gebrauch beiseitegelegt werden?

2.2.5.1 Shapes im Speicher

a) Der Shape-String

Durch SSHAPE wurde ein String geschaffen, der die gleiche Lebensdauer hat wie jede andere Variable: RUN und CLR löschen ihn.

b) Shape im Sprite-Speicher

Wir werden später eine Möglichkeit kennenlernen, die es unter gewissen Umständen erlaubt, ein Shape zum Sprite zu machen. Auf diese Weise gelangt das Muster dazu dann in den Speicherraum, der den Sprites vorbehalten ist, und überlebt, bis der Computer abgeschaltet oder der Speicher überschrieben wird.

2.2.5.2 Shapes auf Diskette oder Kassette

Damit wird's nun wirklich interessant. Der Massenspeicher kann Shapes "auf ewig" festhalten.

a) Shape-Strings direkt

Im Grunde genommen sollte es möglich sein, den durch SSHAPE definierten String direkt als sequentiellen File auf Diskette oder Kassette abzulegen und zu lesen. Allerdings ergeben sich hier - wie auch schon bei den Sprite-Strings - Schwierigkeiten beim Lesen. Weil im Prinzip alle Bytewerte von 0 bis 255 auftreten können, also auch ein Wert, der einem RETURN oder einem Anführungszeichen entspricht, wird der String nicht immer vollständig und fehlerfrei gelesen. Dazu kommt das Problem, daß die Shape-Strings unterschiedliche Längen aufweisen können. Wie bestimmt man das Ende eines solchen Files?

b) Als ASCII-Werte speichern

Ebenfalls bei den Sprites wurde diese Möglichkeit schon angedeutet: den Shape-String auseinanderzunehmen und jedes Byte in Form seines ASCII-Wertes zu speichern. Eine Variante dieses Verfahrens zeigt Ihnen das Programm "SHAPE ERSTELLEN":

Programm SHAPE ERSTELLEN

Shapes retten auf Diskette oder Kassette

```
10 REM***** SHAPE ERSTELLEN *****
20 COLOR0,1:COLOR4,1:COLOR1,6:GRAPHIC1,1:B$=CHR$(13)
30 CIRCLE1,10,10,10,10:SSHAPEA$,0,0,20,20:GOSUB230
40 SCRATCH"SHAPE1"
50 REM***** SHAPES AUF DISKETTE SPEICHERN *****
60 DOPEN#1,"SHAPE1",W:PRINT#1
70 FORI=1TOLEN(A$):PRINT#1,ASC(MID$(A$,I,1))B$:NEXTI:PRINT#1,"ENDE"B$
80 DCLOSE#1
90 REM***** VARIABLE LOESCHEN UND NEUSTART *****
100 CLR:GRAPHIC1,1:SLEEP2:RUN120
110 REM***** SHAPES WIEDER VON DISKETTE HOLEN *****
120 DOPEN#1,"SHAPE1"
130 DO
140 :INPUT#1,B$
150 ::IF B$="ENDE" THEN EXIT
160 :B=VAL(B$):B$=CHR$(B):A$=A$+B$
170 LOOP
180 DCLOSE#1:GOSUB230
190 REM***** SHAPE ZEICHNEN *****
200 GSHAPEA$,100,100
210 END
220 REM***** KONTROLLUNTERPROGRAMM *****
230 FORI=1TOLEN(A$):PRINTASC(MID$(A$,I,1));:NEXTI:PRINT:PRINTLEN(A$):RETURN
```

Hier wird zuerst ein Shape (einfach ein Kreis) erzeugt und in A$ abgelegt. Ein Unterprogramm druckt auf dem Textbildschirm (für Benutzer von 2 Bildschirmen sofort, für andere erst später sichtbar) die ASCII-Werte des Strings aus. In Zeile 40 wird der alte File SHAPE1 gelöscht und dann ab Zeile 50 der neue auf Diskette abgelegt. Benutzen Sie eine Datasette, dann sind lediglich in den Zeilen 60, 80, 120 und 180 die DOPEN und DCLOSE gegen OPEN und CLOSE mit den entsprechenden Gerätenummern auszutauschen. Nun sind wir eigentlich schon fertig: In Zeile 100 löschen wir alle Variablen, den Grafikbildschirm und starten des Programms zweiten Teil, der nur zur Kontrolle das File einliest und unser Shape abbildet. Zusätzlich

werden auf dem Textbildschirm wieder die Stringlänge und die eingelesenen ASCII-Werte abgebildet. Beide Ausdrucke lassen sich gut vergleichen (Benutzer eines Bildschirmes kommen in diesen Genuß, wenn sie nach Ende des Programms durch GRAPHIC0 den Textbildschirm einschalten).

c) Eine unorthodoxe Lösung

Etwas exotisch und daher hier nur kurz skizziert ist die folgende Vorgehensweise: Mittels des POINTER-Befehls (hier also POINTER(A$)) sucht man in BANK 1 den Stringdescriptor. Die dort genannte Adresse und Länge verhelfen uns dazu, den Speicherbereich, in dem wir unseren String stehen haben, durch BSAVE abzuspeichern.

Beim Wiedereinladen müßte man zunächst einen String definieren, dann durch BLOAD dessen Inhalt an eine beliebige Speicherstelle packen und den Stringdescriptor darauf richten.

Das erscheint im ersten Moment ziemlich kompliziert zu sein, könnte aber - besonders bei Verwendung vieler Shapes - schneller funktionieren. Damit würde beispielsweise ein festgelegter Speicherbereich für alle Shapes en bloc ein- oder auslesbar werden. Probieren Sie's doch mal!

2.2.5.3 Shapes für ein Listing

a) Als Zeichenvorschrift

Häufig werden Shapes durch einige wenige Grafik-Befehle erstellt. Die einfachste Methode - der wir uns auch bisher bedient haben in den Beispielprogrammen - ist es daher, die Zeichenvorschrift ins Programm einzuarbeiten. Manchmal stößt man damit aber an Grenzen: wenn beispielsweise der Ausschnitt eines Multicolorbildes als Shape verwendet werden soll, das auf dem Grafiktablett erstellt wurde, oder wenn es sich um einen Teil eines Fractals handelt, das 25 Stunden bis zur Fertigstellung dauert oder ...

b) Shapes in DATA-Zeilen

Ähnlich wie wir bei Sprites einen DATA-Generator verwendet haben, der sich nach seiner Verwendung selbsttätig löscht, geschieht das nun hier auch. Im Programm DATA ZEILEN ist außerdem noch ein kleiner Testteil enthalten.

Programm DATA ZEILEN

Shapes in DATA Zeilen ablegen

```
1 REM ****** ERZEUGUNG VON DATA-ZEILEN AUS SHAPE-STRINGS (HIER A$) ******
2 REM
3 REM ****** PROBESHAPE ERZEUGEN ******
4 REM
5 COLOR0,1:COLOR4,1:COLOR1,6:GRAPHIC1,1:CIRCLE1,10,10,10,10:SSHAPEA$,0,0,20,20
6 STOP
7 REM ****** ERZEUGEN DER DATAZEILEN ******
8 GOTO63000
9 REM ****** NEUES PROGRAMM ZUR KONTROLLE DER DATAS ******
10 COLOR0,1:COLOR4,1:COLOR1,3:GRAPHIC1,1:GOSUB63015
15 GSHAPEC$,100,100
20 END
62999 REM ****** ES FOLGT DER DATAGENERATOR BZW DIE DATAS ******
63000 PRINTCHR$(147)63020"DATA"LEN(A$)
63001 I=1:K=0
63002 DO
63003 :Z=0:PRINT63021+K"DATA";
63004 :DO WHILE I<LEN(A$)
63005 ::PRINTASC(MID$(A$,I,1))",";:I=I+1:Z=Z+1
63006 ::IFZ=10THENEXIT
63007 :LOOP
63008 :PRINTASC(MID$(A$,I,1)):I=I+1:K=K+1
63009 LOOP WHILE I<LEN(A$)
63010 PRINT63021+K"DATA"ASC(MID$(A$,I,1))
63011 PRINT"63015 READA:FORI=1TOA:READB:B$=CHR$(B):C$=C$+B$:NEXTI:RETURN"
63012 PRINT"DELETE63000-63012"
```

Wie gehabt, wird hier zuerst ein Shape (ein Kreis) erzeugt, dann endet das Programm an einem STOP in Zeile 6. Wenn Sie mit dem 40-Zeichen-Bildschirm arbeiten, dann fügen Sie bitte vor das STOP-Kommando noch ein GRAPHIC 0 ein, denn alles, was nun folgt, ist Text. Das STOP-Kommando wurde eingebaut, um Ihnen die Möglichkeit zu geben, an dem Shape vor dem Aufruf des DATA-Generators noch etwas zu verändern. Ist alles in Ordnung, dann geben Sie bitte CONT ein. Der DATA-Generator erzeugt nun 8 Zeilen mit DATAs und dazu noch das Einleseunterprogramm sowie die Löschanweisung DELETE 63000-63012.

Die erste Zahl in den DATAs ist die Stringlänge, die als A später beim Auslesen und Zusammenfügen des Shape-Strings eine wichtige Rolle spielt. Ist das Programm mit dem Ausdrucken der neuen Zeilen fertig, dann bricht es ab. Durch die HOME-Taste gelangen Sie in die erste Zeile. Mittels mehrfach wiederholtem RETURN können nun alle neuen Zeilen ins Programm übernommen und der Generator gelöscht werden. Hinterher sieht unser Programm dann aus wie in DATA ZEILEN (2):

Programm DATA ZEILEN (2)

Das Ergebnis des Programms DATA ZEILEN (2)

```
1 REM ****** ERZEUGUNG VON· DATA-ZEILEN AUS SHAPE-STRINGS (HIER A$) ******
2 REM
3 REM ****** PROBESHAPE ERZEUGEN ******
4 REM
5 COLOR0,1:COLOR4,1:COLOR1,6:GRAPHIC1,1:CIRCLE1,10,10,10,10:SSHAPEA$,0,0,20,20
6 STOP
7 REM ****** ERZEUGEN DER DATAZEILEN ******
8 GOTO63000
9 REM ****** NEUES PROGRAMM ZUR KONTROLLE DER DATAS ******
10 COLOR0,1:COLOR4,1:COLOR1,3:GRAPHIC1,1:GOSUB63015
15 GSHAPEC$,100,100
20 END
62999 REM ****** ES FOLGT DER DATAGENERATOR BZW DIE DATAS ******
63015 READA:FORI=1TOA:READB:B$=CHR$(B):C$=C$+B$:NEXTI:RETURN
63020 DATA 67
63021 DATA 1 , 252 , 0 , 7 , 7 , 0 , 12 , 1 , 128 , 16 , 0
63022 DATA 64 , 32 , 0 , 32 , 96 , 0 , 48 , 64 , 0 , 16 , 192
63023 DATA 0 , 24 , 128 , 0 , 8 , 128 , 0 , 8 , 128 , 0 , 8
63024 DATA 128 , 0 , 8 , 128 , 0 , 8 , 192 , 0 , 24 , 64 , 0
63025 DATA 16 , 96 , 0 , 48 , 32 , 0 , 32 , 16 , 0 , 64 , 12
63026 DATA 1 , 128 , 7 , 7 , 0 , 1 , 252 , 0 , 20 , 0 , 20
63027 DATA 0
```

Zur Kontrolle lassen Sie doch mal dieses Programm durch RUN 10 starten. RUN löscht ja auch die Strings, so daß das nunmehr gezeichnete Shape aus den DATA-Zeilen wieder auferstanden ist.

Die Vorgehensweise bei diesem Programm ist dieselbe wie beim DATA-Generator für die Sprites. Allerdings werden hier die Werte nicht aus dem Speicher, sondern aus einem String geholt, der deshalb auch schon durch einen - zumindest teilweisen - Programmlauf definiert worden sein muß.

Wenn Sie mehrere Shapes auf diese Weise festhalten wollen, ist das Programm relativ einfach umzubauen, am besten mittels eines String-Arrays und einer weiteren Schleife.

Wenden wir uns nun dem Zusammenspiel von Sprites und Shapes zu:

2.3 Shapes und Sprites

2.3.1 Erzeugen von Sprites aus Shapes

Anstatt mit dem SPRDEF-Befehl können Sprites auch regelrecht gezeichnet werden durch Grafik-Befehle im Hochauflösungs- oder im Mehrfarbenmodus. Die Nahtstelle zwischen Sprite und Shape ist dabei der String, in den unsere Zeichnung zuerst als ein Shape (mittels SSHAPE) eingeschrieben wird. Von dort ist es dann durch SPRSAV in den Sprite-Speicher zu übertragen.

Eine Regel ist dabei zu beachten: Der Sprite-String liegt in seiner Länge fest. Im Normalmodus müssen wir immer von einem 24mal21- (das ist X mal Y-) Raster ausgehen, im Mehrfarbenmodus vom 12mal21-Muster. Eine SSHAPE-Anweisung muß daher - falls wir ein Sprite konstruieren wollen - im Normalmodus lauten:

```
SSHAPE A$,X,Y,X+23,Y+20
```

und im Multicolormodus:

```
SSHAPE A$,X,Y,X+11,Y+20
```

X und Y sollen die Koordinaten der linken oberen Ecke des gewünschten Bildausschnittes sein.

Ist SSHAPE nicht in diesem Raster erstellt worden, kann sich allerlei Unsinn auf dem Bildschirm abspielen.

Ein kleines Demoprogrämmchen namens "SHAPES ZU SPRITES" spielt diese Option durch:

Programm SHAPES ZU SPRITES

Shapes werden zu Sprites

```
10 REM ***** SHAPES ZU SPRITES UMWANDELN *****
15 REM********** MULTICOLOR-SPRITES **********
20 COLOR0,1:COLOR1,6:COLOR2,2:COLOR3,3:COLOR4,1:COLOR5,2
30 GRAPHIC3,1:CIRCLE1,5,10,5,10:BOX2,0,0,10,20:DRAW3,0,10TO10,10
40 DRAW3,5,0TO5,20
50 SSHAPEA$,0,0,11,20
60 SPRSAVA$,1:SPRCOLOR6,3:MOVSPR1,100,100:SPRITE1,1,2,0,0,0,1
65 SLEEP2
70 REM********** NORMALER HIRESSPRITE **********
80 GRAPHIC1,1:CIRCLE1,50,10,10,10:CIRCLE1,50,10,7,7:CIRCLE1,50,10,4,4
90 SSHAPEB$,40,0,63,20:GSHAPEA$,0,0
100 SPRSAVB$,2:MOVSPR2,200,200:SPRITE2,1,6
110 MOVSPR1,0#4:MOVSPR2,90#8
```

Zunächst wird ein Multicolorshape erzeugt und in Sprite 1 geschrieben. Eine Umstellung vom Multicolor- in den Hochauflösungsmodus zeigt, daß das Sprite seine Farben behält, das Shape dagegen nicht. Ein zweites Shape erzeugt das Sprite 2, die beide dann über den Bildschirm bewegt werden. Falls Sie nach dem Programmende mittels GRAPHIC 0 noch auf den Textbildschirm (40 Zeichen) umschalten, sehen Sie noch einen Unterschied zwischen Shapes und Sprites: Letztere bleiben auch hier aktiv.

2.3.2 Shapes aus Sprites bauen

Natürlich ist auch der umgekehrte Weg möglich, nämlich Shapes aus Sprites zu konstruieren. Fügen Sie einfach mal an unser letztes Programm "SHAPES ZU SPRITES" folgende Zeilen an:

```
130 CLR :REM LOESCHEN DER STRINGS
150 SPRSAV 1,A$:SPRSAV 2,B$
160 GSHAPE A$,150,100:GSHAPE B$,200,100
```

Über den Sinngehalt solch einer Zuordnung läßt sich natürlich streiten. Denkbar wäre es, das anzuwenden bei Speicherung eines Shape-Musters im Spritespeicher und anschließendem Zurücklesen dieser Information. Darüber hatten wir oben bei Aufbewahrung von Shapes für die Nachwelt schon kurz nachgedacht.

2.3.3 Wenn Shapes und Sprites zusammenstoßen

Kollisionen von Sprites mit Shapes werden vom Computer genauso registriert, als wären es Zusammenstöße mit Textbestandteilen. Sowohl der COLLISION- als auch der BUMP- Befehl können verwendet werden. In einer erweiterten Version unseres Programms SHAPES ZU SPRITES namens "KOLLISION VON SPRITES UND SHAPES" werden alle Möglichkeiten, die wir in den letzten Abschnitten besprochen haben, ausprobiert. Dabei findet der COLLISION-Befehl Anwendung, und Sprite 1 ist durch einen Joystick in Port 2 steuerbar. Bei jedem Zusammenstoß wechselt der Sprite die Farben.

2.3.4 Wann Shapes und wann Sprites?

Jeder von Ihnen kann eigentlich nun die Frage beantworten, wann welches der beiden Grafik-Objekte sinnvoll einzusetzen sei: Shapes oder Sprites.

- Bewegte Objekte sollten Sprites sein
- Objekte, die in verschiedenen Grafikmodi (auch im 40-Zeichen-Textmodus) in gleicher Gestalt auftreten sollen, können nur Sprites sein
- Statische Objekte in beliebiger Anzahl sind als Shapes gut zu verwirklichen
- Objekte, bei denen kein starr festgelegter Raster zugrundegelegt wird, können Shapes sein

Nun sind Ihrer Phantasie keine Grenzen gesetzt: Wollen Sie ein Menü bauen, in dem mit einem Sprite in Pfeil- oder Handform auf Symbole (Shapes) gedeutet wird, oder möchten Sie Schaltpläne aus vorgefertigten Einzelteilen (Shapes) zusammenbauen, die Sie mit einem Kran-Sprite aussuchen und plazieren oder...

Programm KOLLISION VON SPRITES UND SHAPES

Zusammenspiel von Sprites und Shapes

```
10 REM ***** KOLLISION VON SPRITES UND SHAPES *****
20 REM ***** SHAPES ZU SPRITES UMWANDELN *****
30 REM********** MULTICOLOR-SPRITES **********
40 COLOR0,1:COLOR1,6:COLOR2,2:COLOR3,3:COLOR4,1:COLOR5,2
50 GRAPHIC3,1:CIRCLE1,5,10,5,10:BOX2,0,0,10,20:DRAW3,0,10TO10,10
60 DRAW3,5,0TO5,20
70 SSHAPEA$,0,0,11,20
80 SPRSAVA$,1:SPRCOLOR6,3:MOVSPR1,100,100:SPRITE1,1,2,0,0,0,1
90 SLEEP2
100 REM********** NORMALER HIRESSPRITE **********
110 GRAPHIC1,1:CIRCLE1,50,10,10,10:CIRCLE1,50,10,7,7:CIRCLE1,50,10,4,4
120 SSHAPEB$,40,0,63,20:GSHAPEA$,0,0
130 SPRSAVB$,2:MOVSPR2,200,200:SPRITE2,1,6
140 MOVSPR1,0#4:MOVSPR2,90#8
150 SLEEP2
160 REM********** LOESCHEN DER STRINGS **********
170 CLR
180 REM********** UEBERTRAGEN VON SPRITES IN SHAPES **********
190 SPRSAV1,A$:SPRSAV2,B$
200 GSHAPEA$,150,100:GSHAPEB$,200,100
210 REM********** STEUERN VON SPRITE 1 **********
220 CHAR1,20,0,"JOYSTICK PORT 2"
230 REM********** KOLLISION M. SHAPES **********
240 MOVSPR1,0#0
250 X=RSPPOS(1,0):Y=RSPPOS(1,1):I=6:J=3
260 COLLISION2,370
270 ONJOY(2)GOSUB290,300,310,320,330,340,350,360
280 GOTO270
290 Y=Y-2:MOVSPR1,X,Y:RETURN
300 X=X+1:Y=Y-1:MOVSPR1,X,Y:RETURN
310 X=X+2:MOVSPR1,X,Y:RETURN
320 X=X+1:Y=Y+1:MOVSPR1,X,Y:RETURN
330 Y=Y+2:MOVSPR1,X,Y:RETURN
340 X=X-1:Y=Y+1:MOVSPR1,X,Y:RETURN
350 X=X-2:MOVSPR1,X,Y:RETURN
360 X=X-1:Y=Y-1:MOVSPR1,X,Y:RETURN
370 COLLISION2
380 I=I+1:IFI>16THENI=1
390 J=J+3:IFJ>16THENJ=1
400 SPRCOLORI,J
410 COLLISION2,370
420 RETURN
```

3 Schönheit im Chaos

Als Beispiel für die Anwendung von Mehrfarbengrafik auf unserem Commodore 128 behandeln wir ein Thema, das in den letzten Jahren zunehmend an Bedeutung gewonnen hat, die sogenannten Fractals. Durch Rekursion (dieser Begriff wird gleich noch erklärt) meist sehr einfacher mathematischer Beziehungen lassen sich mit Hilfe des Computers Bilder von überraschender Schönheit erzeugen. Der Computer und die neuesten Erkenntnisse der angewandten Mathematik als Instrumente der bildenden Kunst? Versuchen Sie es selbst!

3.1 Die Realität ist fractal.

Bevor wir uns die Praxis ansehen - Rezepte dafür sind häufig zu finden (1) (6) - sollten wir uns etwas mit der Bedeutung der Fractals befassen. Daß solch ein Gebilde nämlich als Computerkunst unsere ästhetische Ader anspricht, ist eigentlich nicht mehr als ein erfreulicher Nebeneffekt. In Wahrheit sind diese hübschen Bilder ein Ausdruck des derzeitigen Wandels im naturwissenschaftlichen Weltbild, der von vielen Zeitgenossen geradezu als Revolution empfunden wird. Das erfordert eine Erklärung:

Prinzipiell existieren in der Natur zwei gegenläufige Tendenzen: eine zerstreuende (dissipative) und eine ordnende. Letztere findet ihren Ausdruck beispielsweise in der Regelmäßigkeit von Kristallen oder Planetensystemen. Die klassische Physik versucht seit Jahrtausenden alles Weltgeschehen von diesem Denkansatz (die Ordnung der Dinge zu ergründen) her zu erklären. Durch Idealisieren und Abstrahieren wurden Theorie und Experiment zum mächtigen Gebäude der Naturwissenschaften zusammengefügt.

Relativ neu im Vergleich dazu ist die Untersuchung der anderen, der zerstreuenden Tendenz. Sie findet ihren ersten Ausdruck in der sogenannten Molekularkinetik (Stichworte dazu sind Entropie und Wärmetod) und ihren derzeitigen Höhepunkt in der modernen Quantentheorie. Hier arbeitet der Naturwissenschaftler mit dem Instrumentarium der Statistik und Wahrscheinlichkeitsrechnung. Vielerorts herrscht aber die Meinung, daß man - wenn die Materie bis in die grundlegenden Bausteine erkannt sei - prinzipiell auch diese Bereiche der Natur mit den Mitteln einer erweiterten klassischen Physik beschreiben kann (das Wort "klassisch" ist hier etwas gewagt, weil zu dieser Art Physik auch die Erkenntnisse der Relativitätstheorie gehören würden).

Eine Basis des naturwissenschaftlichen Denkens ist der sogenannte Determinismus: Jede Wirkung hat ihre Ursache. Der Glaube, daß man - kennt man nur alle Einflußgrößen und physikalischen Zusammenhänge und verfügt man über ausreichende Rechenkapazität - jedes Ereignis vorherberechnen kann, ist so alt wie unsere Kultur (daher der Gebrauch des Wortes "klassisch", denn auch die Relativitätstheorie bricht nicht mit dem Glauben an Ursache-Wirkungs-Ketten). Diese Basis ist in jüngster Zeit erschüttert worden durch die Erkenntnisse zweier Arbeitsgebiete der modernen Naturwissenschaft, die auf den ersten Blick nicht viel miteinander zu tun haben.

Der Begriff der "Ursache" einer Wirkung wird in der modernen Elementarteilchenphysik (in der S-Matrix-Theorie) in Frage gestellt (5). Von der anderen Seite sägt B. Mandelbrot am Fundament des Determinismus: Er arbeitet am Thema "Extreme und unvorhersagbare Unregelmäßigkeiten von Naturphänomenen in Physik, Soziologie und Biologie". Gleichzeitig ein Mittel zur Beschreibung und ein Ausdruck solcher Unregelmäßigkeiten in der Natur sind die Fractals. Mit der Verleihung der Barnard-Medaille 1985 durch die Nationale Akademie der Wissenschaften der Vereinigten Staaten ist die Bedeutung seiner Arbeiten für die Wissenschaft deutlich geworden. Frühere Preisträger sind beispielsweise Albert Einstein, Niels Bohr und Werner Heisenberg (4).

3.2 Was sind Fractals?

Drei Ansätze zum Verständnis von Fractals werden wir nun kennenlernen, einen geometrischen, einen, der mit der Dynamik von Wachstumsprozessen zusammenhängt, und schließlich noch einen mathematischen.

3.2.1 Fractals und Geometrie

Mandelbrot hat einmal erklärt: "Die Natur hat der klassischen Geometrie ein Schnippchen geschlagen. Denn Wolken sind eben keine Kugeln, Berge keine Kegel, Inseln keine Kreise und Baumstämme keine Zylinder. Ebenso bewegt sich ein Blitz nicht auf einer Geraden. Und die Oberfläche der Planeten ist nicht glatt." (Zitiert in (4)). Es ergibt sich die Notwendigkeit, eine neue Geometrie anzuwenden, eine Geometrie der Natur.

Objekte dieser Geometrie werden durch rekursive Methoden (wir kommen wirklich bald zur Erklärung, was das eigentlich ist) gewonnen. Sie haben merkwürdige Eigenschaften. So gibt es Kurven, die eine kleine Fläche umranden und dennoch unendlich lang sind. Die Unendlichkeit drückt sich darin aus, daß man bei immer genauerem Hinsehen immer kompliziertere Strukturen erkennt. Der Begriff der Dimension (Erinnern Sie sich: Ein Punkt ist ein Gebilde mit 0 Dimensionen- er hat keine Ausdehnung. Eine Linie gehört zur ersten Dimension, ihre Ausdehnung läßt sich als Länge angeben. Eine Fläche ist zweidimensional, weil wir von Länge und Breite reden können. Körper sind dreidimensionale Objekte, bei denen noch die Höhe eine Rolle spielt.), dieser Begriff der Dimensionen also gerät ins Wanken: Die Objekte der neuen Geometrie sind beispielsweise nicht mehr eindimensional, aber noch nicht zweidimensional (oder aber nicht mehr zwei- und noch nicht dreidimensional). Sie haben eine gebrochene, nichtganzzahlige Dimensionalität. Daher der Name Fractal, vom lateinischen "frangere", was brechen heißt.

Ein Beispiel für fractale Geometrie nennt Fricker (4): Ein britisches Forschungsteam hat die Oberfläche von gewissen Pflanzen untersucht und dafür die (fractale) Dimension 2.79 bestimmt. Die Folgerung daraus klärt einen Widerspruch: Werden diese Pflanzen von zehnfach kleineren Insekten als ursprünglich bevölkert, dann nimmt deren Zahl nicht - wie man erwarten sollte - um das Hundertfache, sondern um etwa das Sechshundertfache zu. Den Grund dafür erklärt die Rechnung mit Fractals: 10^2.79 = 617, anstelle von 10^2 = 100.

3.2.2 Fractals und Wachstumsdynamik

Die Gesetze des exponentiellen Wachstums sind schon seit langem bekannt. Am Beispiel der Bevölkerungsexplosion sollen sie kurz erklärt werden. Wenn X(n) die Bevölkerung im Jahr n ist und p die jährliche Zuwachsrate, dann ergibt sich für die Bevölkerung im Jahr n+1 die Beziehung

$$X(n+1) = (1+p)*X(n)$$

Will man die Bevölkerungszahl im Jahr n+2 erfahren, dann setzt man in diese Gleichung anstelle von X(n) nun X(n+1) ein. Für das Jahr n+3 setzt man X(n+2) ein und so fort. Es ergibt sich eine immer steiler steigende Kurve, wenn man die Bevölkerungszahlen X(i) gegen die jeweiligen Jahreszahlen i grafisch darstellt: die Bevölkerungsexplosion.

Was wir bei diesem Beispiel gemacht haben, nennt man Rekursion. Das Ergebnis einer Berechnung wurde jedesmal wieder zurückgeführt in die Gleichung. "recurrere" ist lateinisch und bedeutet "zurücklaufen".

Nun ist es nicht erst seit dem Bericht des Club of Rome bekannt, daß gewisse Einflußfaktoren - beispielsweise ein Mangel an Rohstoffen oder Nahrung - dem Wachstum Grenzen setzen. Ein belgischer Naturwissenschaftler namens Verhulst hat schon in der Mitte des vorigen Jahrhunderts ein Wachstumsgesetz entdeckt, das einen Ausdruck

$$p*(1-a*X(n))$$

verwendete. In völlig anderen Zusammenhängen spielt dieser Ausdruck in der modernen Wissenschaft eine wichtige Rolle (Laserphysik, Evolutionstheorie, turbulente Strömung von Medien). Entwickelt man diese Beziehung rekursiv, dann kann - in Abhängigkeit von den jeweiligen Parametern und nach genügend häufigen Schritten - dreierlei geschehen:

- Exponentielles Wachstum (wie schon gehabt)
- Zurückgehen der Ergebnisse bis auf Null oder einen jeweiligen Grenzwert
- Dazwischen aber gibt es merkwürdige Verhaltensweisen. Man kann bei bestimmten Ausgangswerten ein Oszillieren der Ergebnisse beobachten, das mehr oder weniger regelmäßig erfolgt (3) und stark von dem exakten Wert der Wachstumsrate abhängt. Schon eine klitzekleine Änderung der Ausgangsbedingungen kann entscheiden über einen völlig anderen Verlauf der Wachstumskurven.

Dieses Grenzdasein ist untrennbar mit Fractals verbunden. Auf welche Weise, das soll uns der nächste Abschnitt zeigen. Interessant in dem Zusammenhang ist, daß die bisher erwähnten Anspielungen auf Fractals ebenfalls immer Grenzen betrafen, die Oberfläche einer Pflanze, die Umrandung einer Fläche, die schmale Linie zwischen exponentiellem Wachstum einerseits und Abfallen auf Null andererseits...

3.2.3 Ein einfacher mathematischer Weg zu Fractals

B. Mandelbrot (9,10) hat eine Anzahl Funktionen untersucht, von denen eine der vorhin gezeigten Wachstumsfunktion ähnelt:

X(n+1) = X(n)^2+C

Wir nehmen also einen beliebigen Wert X(n), quadrieren ihn und addieren einen konstanten Wert C. Wieder setzen wir das Ergebnis X(n+1) als X(n) neu ein und beobachten, wie sich bei fortgeführter Iteration (also weiteren Rekursionen) der berechnete Wert verhält.

Wenn C = 0 ist, ist die Situation relativ einfach

Falls wir mit einem X-Wert kleiner als 1 beginnen, ergibt sich durch Quadrieren ein noch kleinerer Wert, und bei weiterer Rekursion nähert sich das Ergebnis schließlich der Null.

Wenn wir dagegen einen Startwert größer als 1 wählen, dann ist sein Quadrat noch größer, und sehr schnell strebt das Ergebnis gegen Unendlich.

Man nennt diese Zahlen, gegen die jeweils die Ergebnisse tendieren, "Attraktoren" (3). So haben wir im bisher betrachteten Fall die Attraktoren 0 und Unendlich. Jeder dieser Attraktoren besitzt Einflußsphären: Zu 0 gehören alle X-Anfangswerte, deren Absolutbetrag kleiner als 1 ist, zu Unendlich gehören alle anderen Anfangswerte. Die Zahlen 1 und -1 bilden Grenzen der Einflußsphären.

Wählen wir nun einen anderen Wert für die Konstante C, beispielsweise C=1. Beginnt man nun mit X=0, dann erhält man der Reihe nach

0, 1, 2, 5, 26, 677,...

Welchen Startwert für X wir auch immer wählen (probieren Sie es mal aus!), jedesmal entwickelt sich das Ergebnis nach wenigen Schritten zur Unendlichkeit hin. Ein Attraktor ist verlorengegangen!

Nehmen wir nun mal den Wert -1 für die Konstante C und starten mit X=1, dann finden wir diese Ergebnisreihe:

1, 0, -1, 0,- 1, 0, -1, 0, ...

Die Werte pendeln hin und her um einen periodischen Attraktor, der seinen Einflußbereich auf die X-Startwerte zwischen -1.618 und +1.618 ausdehnt. Versuchen wir beispielsweise den Startwert .5, dann ergeben sich

.5, -.75, -.4375, -.8086, -.346, -.880, -.2253, usw.

Alle anderen Startwerte tendieren schnell zur Unendlichkeit, beispielsweise beim Startwert 3:

3, 8, 63, 3968, ...

Wir haben demnach nun wieder 2 Attraktoren, von denen einer ein periodischer ist und der andere die Unendlichkeit.

Probieren Sie dieses Spiel noch etwas durch mit verschiedenen Konstanten C und Startwerten X. Sie werden immer finden, daß ein Attraktor unendlich ist. Für C-Werte zwischen -2 und .25 gibt es dann noch einen 2. Attraktor, der entweder ein einfacher oder ein periodischer ist oder aber ein merkwürdig chaotischer (3).

Wir sehen schon, daß mit dieser einfachen mathematischen Gleichung allerhand Interessantes anzustellen ist. Um nun zur grafischen Darstellung dieser Zusammenhänge zu gelangen - denn das sind die Fractals, wie Sie sie in Abbildungen sehen -, werden wir noch eine besondere Art von Zahlen kennenlernen, die uns wesentlich mehr Möglichkeiten bietet.

3.2.4 Komplexe Zahlen

Keine Angst, es wird jetzt nicht plötzlich abgehoben in die höheren Sphären der Mathematik. Komplexe Zahlen sind nämlich im Grunde genommen ganz einfach zu verstehen. Ihre Einführung ergab sich aus der Wurzelrechnung. Eine Quadratwurzel wie SQR(4) ist ja definiert als die

Zahl, die mit sich selbst multipliziert das Argument der Wurzel ergibt. In unserem Beispiel ist das Argument die 4 und 2*2 ist 4, somit ist die Quadratwurzel von 4 die 2. Weitere Beispiele:

3 = SQR(9)
1 = SQR(1) aber auch
-3 = SQR(9), weil auch (-3)*(-3)=9 gilt.

Probleme werden Sie haben, wenn Sie auf Ihrem Computer mal versuchen, die Aufgabe

```
PRINT SQR(-2)
```

zu lösen. Es gibt keine Zahl aus dem normalen Zahlenbereich, die mit sich selbst malgenommen eine negative Zahl ergibt. Der Computer steigt mit einer Fehlermeldung aus. Auch die Menschen sind bei dieser Fragestellung lange Zeit "ausgestiegen". Irgendwann (das war im 17.Jahrhundert) kam aber mal jemand auf die Idee, eine neue Sorte von Zahlen zu erfinden, die sogenannten imaginären Zahlen, deren einfachstes Glied die Zahl i ist. Dabei sollte i=SQR(-1) sein. Man kann tatsächlich damit rechnen, und manches Mal ergeben sich aus solchen Rechnungen wieder unsere gewohnten Zahlen, wenn nämlich einmal i^2 auftritt, was -1 ist:

i = SQR(-1)
i^2 = -1

Noch etwas komplizierter wird das mit den sogenannten komplexen Zahlen. Die sind zusammengesetzt aus einem Realteil (das sind unsere normalen Zahlen) und einem Imaginärteil (also einem Teil, in dem i eine Rolle spielt). Solche Zahlen sehen immer so aus:

z = a + b*i
Realteil Imaginärteil

Mit solchen komplexen Zahlen kann man auch wunderbar rechnen, was Ihnen an der Addition gezeigt werden soll:

(2+3i) + (4-i) = 6 +2i

Sie sehen, man behandelt einfach beide Teile gesondert (2+4=6 berechnet die Summe des Realteils und 3i-i=2i die des Imaginärteils). Alle Rechenarten sind jedenfalls möglich, nur wir werden sie hier nicht zeigen.

Zwei Begriffe spielen für unsere weiteren Absichten noch eine Rolle. Zunächst dreht es sich dabei um den Betrag einer komplexen Zahl. Wenn wir von der komplexen Zahl

z = a + bi ausgehen, dann ist
/z/ = SQR(a^2 + b^2) ihr Betrag.

Zum zweiten soll noch eine Methode zur grafischen Darstellung komplexer Zahlen vorgestellt werden, die der berühmte Mathematiker Gauß entwickelt hat: Die Gaußsche Zahlenebene. In Bild 3.1 ist das Prinzip erläutert:

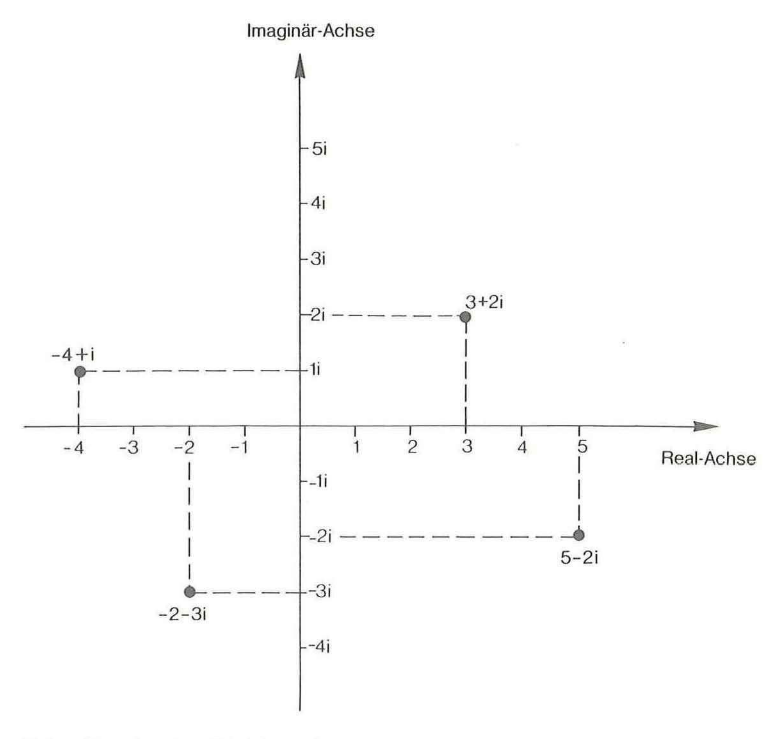


Bild 3.1: Die Gaußsche Zahlenebene

Auf den ersten Blick sieht das aus wie ein normales Koordinatensystem. Bei näherem Hinsehen erkennen Sie, daß in der horizontalen Achse der Realteil, in der vertikalen aber der Imaginärteil einer komplexen Zahl erfaßt wird. Jeder komplexen Zahl entspricht nun ein Punkt auf dieser Ebene.

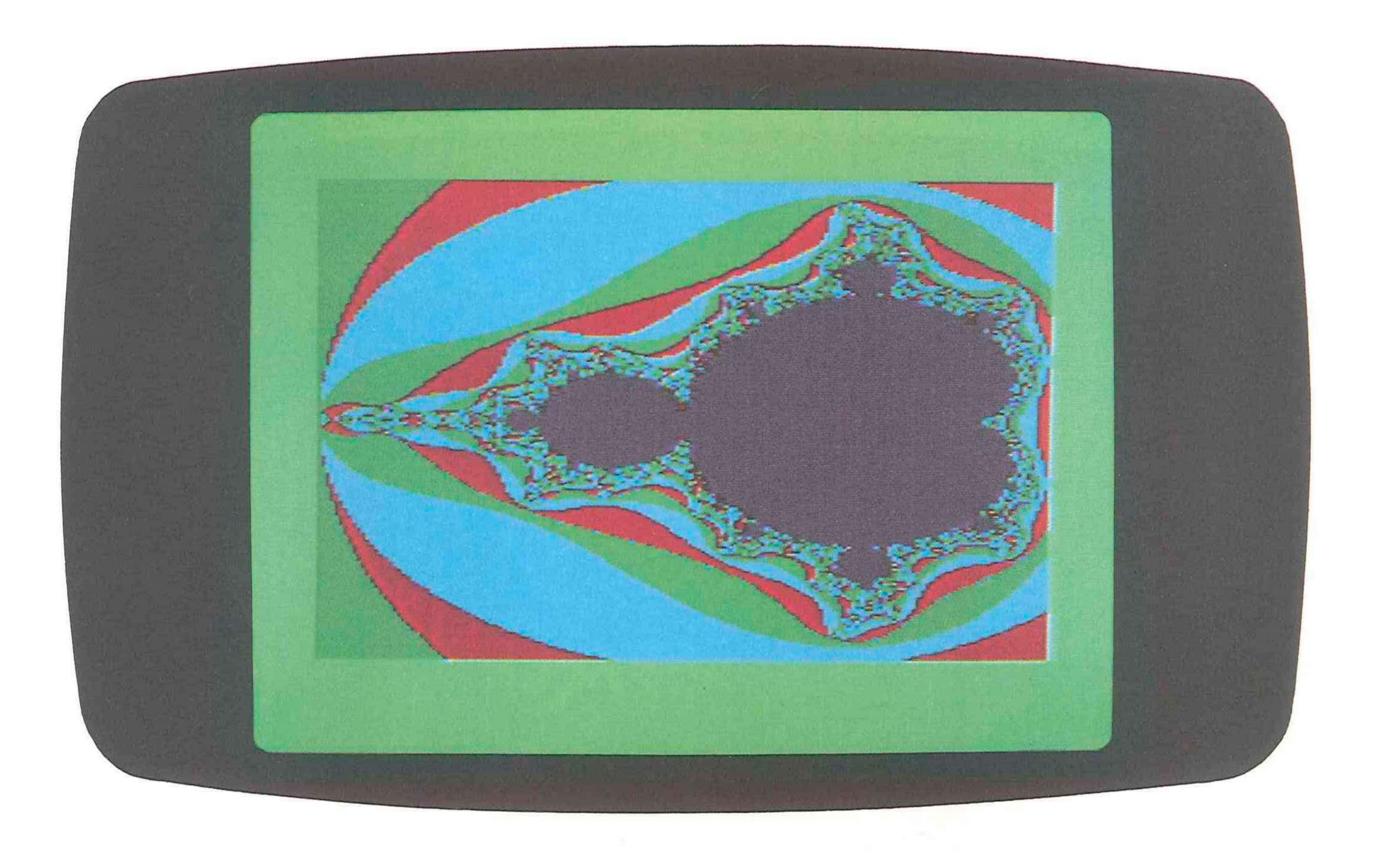


Bild 3.2: Die normale Mandelbrot-Menge: Das "Apfelmännchen"



Bild 3.4: Fractal in zwei Auflösungen



Bild 3.5: Ein leicht verzerrtes "Apfelmännchen"

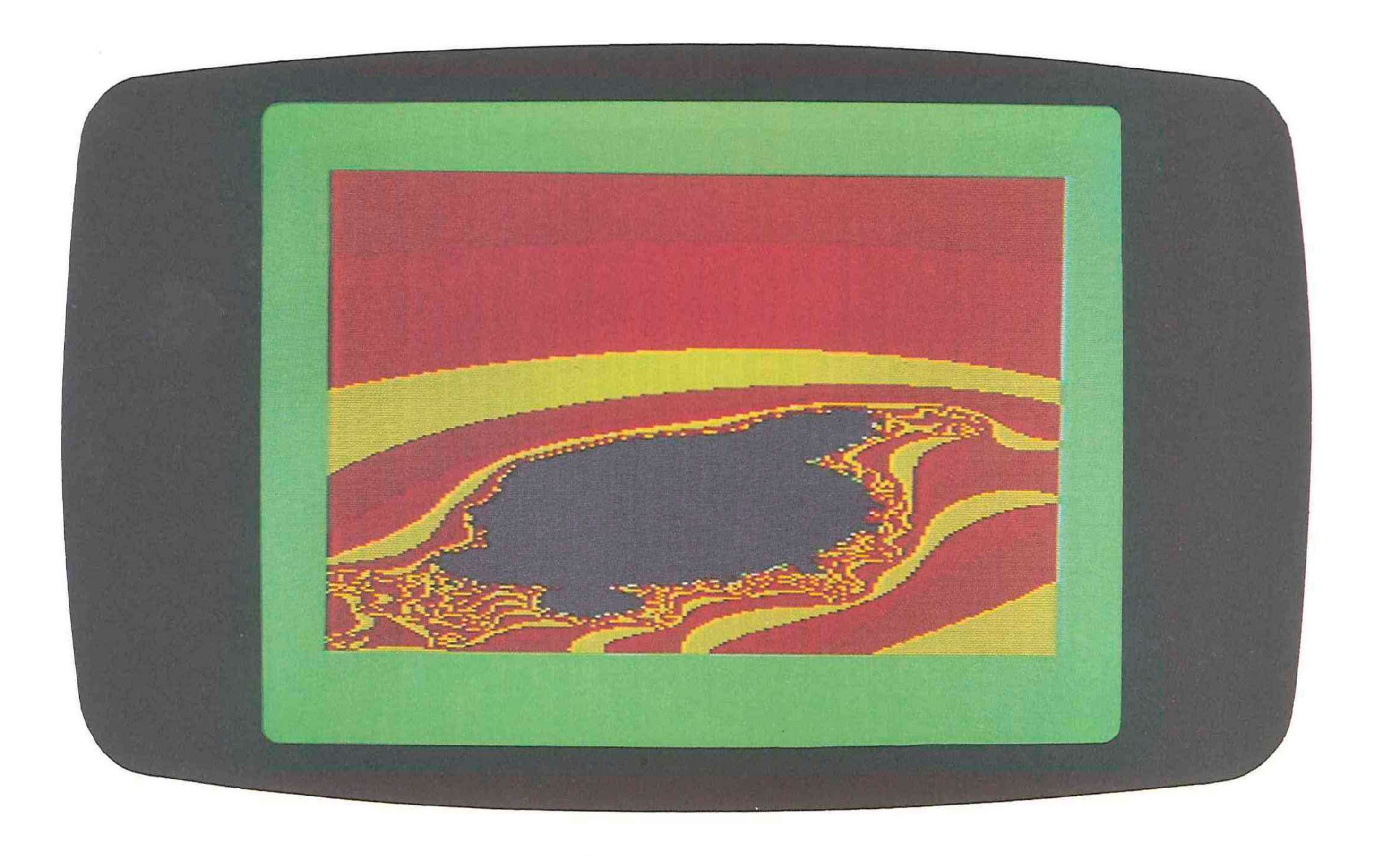


Bild 3.6: Ein stark verzerrtes "Apfelmännchen"

Noch eine kleine Bemerkung zur sicher bei Ihnen auftretenden Frage, was man denn überhaupt mit diesen merkwürdigen Zahlen anfangen kann. Fragen Sie doch mal einen Ingenieur oder einen Elektrotechniker oder einen Physiker oder einen ...

Damit haben wir nun fast alles erforderliche Rüstzeug, um Fractals erzeugen zu können.

3.2.5 Die Mandelbrot-Menge

Erinnern wir uns an die Gleichungen, die wir vorhin für eine mathematische Einführung in die Fractals gebraucht haben. Auch hier verwenden wir nun solch eine Beziehung:

m = z^2 +c

Hier ist allerdings z eine komplexe Variable (wir verwenden im folgenden dafür z=x+yi) und c eine komplexe Konstante (im folgenden ist dann c=a+bi).

Ausführlich geschrieben haben wir dann die Gleichung:

m = (x+yi)^2 + (a+bi)

Auch hier arbeiten wir rekursiv, setzen also jedes Ergebnis wieder als neues z in die Gleichung ein.

Wir beginnen mit z=0, wobei dann x und y gleich 0 sind. Setzt man das in die Gleichung ein, dann erhält man

m1= z^2 +c = c

Der zweite Schritt setzt anstelle von z nun c ein: m2= c^2 +c. Nun wird die Sache interessant. Im 3. Schritt folgt nämlich:

m3= (c^2+c)^2 +c

Sie sehen jetzt zweierlei: Zum einen wird die Angelegenheit im weiteren Verlauf schnell unübersehbar, und zum zweiten können Sie sich sicher vorstellen, daß - je nach dem Wert von c - schnell große Werte als Ergebnis auftreten werden. Jedes Ergebnis m kann man als Punkt auf der Gauß-

Ebene darstellen. Die meisten Zahlen für c allerdings werden bald zu Ergebnissen führen, die aus jedem im noch so großen Maßstab gezeichneten Gauß-System heraus ihren Weg zur Unendlichkeit antreten.

Probieren wir das mal mit einem c aus (2): c = 1 + 1i. In der Reihenfolge der Ergebnisse finden sich dann:

m1 =	1 +	1i
m2 =	1 +	3i
m3 =	-7 +	7i
m4 =	1 -	97i
m5 =	-9407 -	193i
m6 =	88454401 +	3631103i

Allerdings steigen nicht bei allen C-Werten die Ergebnisse über alle Grenzen. Einige existieren, bei denen auch beliebig häufige Iteration immer noch zu endlichen Zahlen führt. Die Menge dieser C-Werte nennt man die Mandelbrot-Menge nach ihrem Entdecker B. Mandelbrot. Sie ergibt in der grafischen Darstellung das "Apfelmännchen" (Bild 3.2 - Siehe Farbteil I, Die normale Mandelbrot-Menge: Das "Apfelmännchen").

Wenn also der Realteil von C etwa zwischen -2 und +0.5, der Imaginärteil etwa zwischen -1.25 und +1.25 liegt, dann hat man gute Chancen, auch nach extrem vielen Iterations-Schritten noch endliche Ergebnisse zu erhalten, denn C liegt dann innerhalb der Mandelbrot-Menge.

Ist C so gewählt, daß es außerhalb dieses Bereiches darzustellen ist (die Darstellungsebene ist eine Gauß-Zahlenebene!), dann streben die Ergebnisse schon nach mehr oder weniger Iterationen gegen Unendlich.

Die Trennzone aber zwischen diesen beiden Möglichkeiten hat fractalen Charakter. Dazu werden wir gleich noch kommen. Wir wollen uns nun die ganze Materie unter programmtechnischen Gesichtspunkten ansehen.

3.2.6 Programm zum Abschätzen der Mandelbrot-Menge

Dewdney (2) erwähnt, daß die Iterationstheorie ein untrüglicher Maßstab sei, um zu erkennen, wann eine Rekursion zum Attraktor Unendlich führt. Mit Hilfe des Betrages unserer Ergebnisse ist das möglich: Wenn nämlich dieser Betrag irgendwann einmal größer als 2 wird, dann treibt bald danach auch das Ergebnis gegen Unendlich.

Entwickeln wir also zunächst ein kleines Programm, das uns die Ergebnisse der Iterationen und ihren Betrag - das ist nämlich die Entfernung vom Nullpunkt des Koordinatensystems, in dem wir die Mandelbrot-Menge gesehen haben - berechnen hilft. Dazu bringen wir zuerst einmal unsere Ausgangsgleichung in eine computergerechte Form:

Diese Gleichung hieß ja

m = z^2 + c

Und dabei waren die Zahlen z und c komplex:

z = x + yi
c = a + bi

Ausgeschrieben hieß dann unsere Gleichung

m = (x+yi)^2 + (a+bi)

Daraus folgt durch Umformung:

m = (x^2 - y^2 + a) + (2xy + b)i
Realteil Imaginärteil

Wir definieren nun 2 Funktionen. Für die Berechnung des jeweiligen Realteils ist das

```
FN R(V) = X*X - Y*Y + A
```

und für die Berechnung des Imaginärteils

```
FN I(V) = 2*X*Y + B
```

Jede neue Iteration erfordert nun nur noch die Einsetzung des Ergebnisses in z, also die Zuweisungen:

R = FN R(V):X = R
I = FN I(V):Y = I

Dabei ist übrigens V lediglich eine Dummy-Variable. Zur Berechnung des Betrages brauchen wir noch die Beziehung:

D = SQR(R^2 + I^2)

Das beigefügte Programm "MANDELBROT1" übernimmt all diese Aufgaben und druckt Ihnen auf dem Bildschirm folgende Angaben aus: Iterationszahl, Real- und Imaginärteil des jeweiligen Ergebnisses und schließlich den Betrag.

Programm MANDELBROT1

Ein Programm zum Abschätzen der Mandelbrot-Menge

```
10 REM *** PROGRAMM MANDELBROT 1 ***
20 TRAP190
30 X=0:Y=0:A=0:B=0:N=0:R=0:I=0:D=0:V=0
40 DEF FN R(V) = X*X-Y*Y+A
50 DEF FN I(V) = 2*X*Y+B
60 PRINT CHR$(147) CHR$(17) "MANDELBROT-ALGORITHMUS"
70 PRINT CHR$(17) CHR$(17) "M = (X+Y*I)↑2 + (A+B*I)" CHR$(17)
80 INPUT"A,B";A,B:INPUT"X,Y";X,Y
90 PRINT CHR$(147) "ITERATION","REALTEIL","IMAG.TEIL","BETRAG":PRINT
100 R = FN R(V):I = FN I(V)
110 PRINTN,R,I
120 DO
130 :X=R:Y=I
140 :R = FN R(V):I = FN I(V):N=N+1:D=SQR(R*R+I*I)
150 :PRINTN,R,I,D
160 ::GETKEYA$:IF A$="←" THEN EXIT
170 LOOP
180 END
190 IF ER = 15 THEN PRINT"ERGEBNIS IST AUF DEM WEG ZUR UNENDLICHKEIT"
200 END

READY.
```

Probieren Sie nun mal eine Reihe von Kombinationen durch. Für den Anfang rate ich Ihnen, für X und Y 0 einzusetzen. Sie bewegen sich dann nämlich im System der Mandelbrot-Menge, wie es Bild 3.2 zeigt.

Beim Ausprobieren finden Sie Punkte für C, die sehr schnell zu einem Betrag größer als 2 führen. Das sind diejenigen, die sich außerhalb des "Apfelmännchens" befinden. Andere C-Werte brauchen anscheinend unendlich lange dazu. Hier haben wir dann mit einiger Sicherheit Elemente der Mandelbrot-Menge vorliegen. Dazwischen aber liegen Bereiche für C, bei denen der Betrag lange Zeit braucht, bis er größer als 2 wird. Das ist wieder ein Grenzbereich, der fractalen Charakter aufweist. Wie fractaler Charakter aussehen kann, wird uns nun das nächste Programm zeigen.

3.3 Das Zeichnen von Fractals

Das Ausprobieren mit dem Programm MANDELBROT1 könnte uns auch der Computer abnehmen. Wir schreiben ein Programm MANDELBROT2, das alle Kombinationen für C (also die Kombinationen für den Realteil und den Imaginärteil) innerhalb gegebener Grenzen selbst ausführt.

Dazu brauchen wir 4 Eingaben, nämlich die Unter- und die Obergrenzen des Realteils sowie des Imaginärteils, innerhalb derer C variiert werden soll. Dann müssen wir uns eine sinnvolle Form der Ergebnisausgabe überlegen. Wir wählen - wie Sie sich nun schon denken können - eine grafische Ausgabe, die unsere Multicolorgrafikfähigkeit ausnutzt.

Auf dem Bildschirm wird ein Koordinatensystem definiert, dessen Y-Achse die Imaginärachse einer Gauß-Ebene ist (und vom kleinsten zum größten Imaginärteil von C reicht). Die X-Achse soll die Variation des Realteils von C ausdrücken und geht daher vom vorher eingegebenen unterem zum oberen Realteil von C. Alle Punkte dieser Gauß-Ebene entsprechen dann verschiedenen Konstanten C (siehe Bild 3.3).

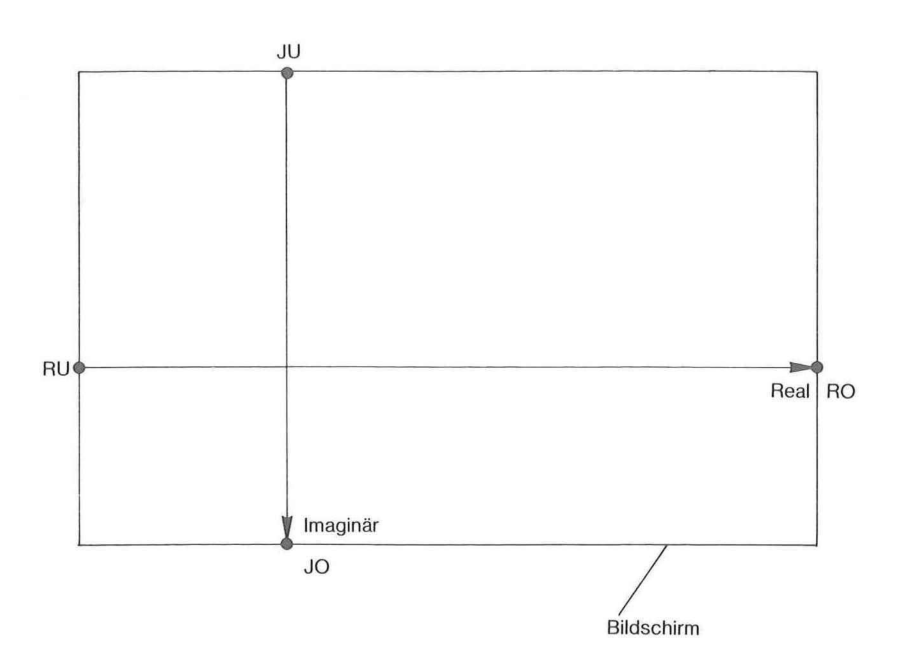


Bild 3.3: Die Gauß-Ebene im Programm Mandelbrot 2

Das Programm MANDELBROT2 führt nun für jeden Wert C die Iterationen durch und prüft dabei jedesmal, ob der Betrag des Ergebnisses schon 2 überschreitet. Ist das der Fall, merkt es sich die Anzahl der Schritte bis dahin. Andernfalls folgen weitere Rekursionen bis zu einer vorher angegebenen maximalen Iterationsanzahl. Ist bis zu dieser höchsten Iterationszahl der Betrag immer noch nicht 2, dann wird das dazugehörige C als schwarzer Punkt eingezeichnet. Dieser Wert C liegt dann in der Mandelbrot-Menge. Alle anderen C werden in einer Farbe gezeichnet, die mit der Anzahl der Iterationen zusammenhängt.

Hätten wir bei einer maximalen Iterationszahl von 100 auch 100 verschiedene Farben zur Verfügung, dann wäre unser Problem schon gelöst. Auf diese Art werden beispielsweise auf größeren Grafik-Computern die Bilder von Fractals erzeugt. Wir sind da leider etwas eingeschränkt auf nur 4 Farben, von denen wir eine (zum Beispiel Schwarz) schon für die Punkte aus der Mandelbrot-Menge vergeben haben. Um eine sichtbare Struktur zu erzeugen, wiederholen wir in einer bestimmten Reihenfolge die verbleibenden 3 Farben. Wir bedienen uns dazu einer Funktion, die leider nicht zum BASIC-Wortschatz unseres BASIC 7.0 gehört, der modulo-Funktion.

Wie wirkt diese Funktion? Sehen wir uns das am Beispiel von modulo(3) einmal an. Das Ergebnis ist immer der Rest, der bei Division durch 3 übrigbleibt:

Zahl	modulo(3) davon
3	0
4	1
5	2
6	0
7	1 usw.

Wir können uns solch eine modulo-Funktion definieren durch

DEF FN MD(X) = X - INT(X/D)*D

Hier ist D der Divisor, in unserem Beispiel also die Zahl 3. X ist dann die Zahl, mit der die modulo-Funktion arbeiten soll. In Zeile 4 des Programms finden Sie diese Funktion, die in Zeile 310 zum Zeichnen benutzt wird.

Einige weitere Erklärungen zum Programm:

Die Zeilen 1 bis 4 enthalten alle benötigten Variablen. Sie sind hier so angeordnet, daß die am häufigsten gebrauchten möglichst weit vorne stehen. Der BASIC-Interpreter findet sie dann schneller. In 120 bis 160 werden alle Eingaben verlangt:

RU,RO = unterer und oberer Rand des Realteils von C. Innerhalb dieser Grenzen wird dann C variiert.

IU,IO = dasselbe für den Imaginärteil von C.

NMAX = Das ist die maximale Iterationszahl. Man muß bedenken, daß von dieser Zahl die Zeichendauer ganz entscheidend abhängt. Im schlimmsten Fall müssen 160*200*NMAX Iterationen ausgeführt werden, bei NMAX = 100 also schon 3,200,000 Schleifendurchläufe! Das wäre allerdings nur dann der Fall, wenn die gesamte Zeichenebene innerhalb der Mandelbrot-Menge läge.

F1,F2,F3,F4 = Das sind die Farben, in denen unsere Abbildung erstrahlen soll. Dabei ist F1 die Farbe, die mit COLOR3 ins Mehrfarbenregister geschrieben wird,

F2 wird mit COLOR2 in das andere Mehrfarbenregister geschrieben,

F3 landet mittels COLOR1 in dem Register für die aktuelle Vordergrundfarbe und

F4 bestimmt den Rand und die Hintergrundfarbe. In der Farbe F4 wird dann auch die Mandelbrot-Menge selbst gezeichnet.

Nach all diesen Eingaben wird in den FAST-Modus geschaltet (sollten Sie zwei Bildschirme verwenden, dann können Sie in den Zeilen 110 und 400 statt GRAPHIC0 ein GRAPHIC5 einsetzen und so den Textbetrieb weiter verfolgen). Zeile 180 berechnet die Schrittweiten der Variation des Real- und des Imaginärteils von C. Es folgt eine Dreifachschleife: Ganz außen wird die Vertikale (also der Imaginärteil) durch die Variable U durchgezählt. Innerhalb dieser Schleife liegt die Schleife für die Horizontale mit dem Schleifenzähler V. Die eigentliche Iteration geschieht in der inneren DO...LOOP-Schleife. Zwei Ausstiegsbedingungen sind vorgegeben: In Zeile 250 wird der Betrag des Ergebnisses als Quadrat (wegen der schnelleren Rechnung) abgefragt. Übersteigt dieser den Wert 2^2=4, dann ist die Schleife beendet. Die andere Abbruchbedingung befindet sich in Zeile 290, wo die Anzahl der Iterationen verglichen wird mit der maximalen (vorher eingegebenen) Schrittzahl. Je nach dem Ausstieg aus dieser inneren Schleife wird dann in Zeile 310 ein Punkt gezeichnet. Als Farbquelle dient das durch modulo(3) errechnete Register.

Der Rest des Programms dient zum Abspeichern des errechneten Bildes.

Programm MANDELBROT2

Ein Programm zum Zeichnen von Fractals

```
1 N%=0:NM%=0:X=0:Y=0:R=0:I=0:D=0:V=0:U=0:A=0:B=0:DA=0:DB=0
2 IU=0:IO=0:RU=0:RO=0:F1%=1:F2%=1:F3%=1:F4%=1:XR=0:YI=0
3 A$="":B$=""
4 DEFFNMD(X)=X-INT(X/3)*3
10 REM ********************************
20 REM *                              *
30 REM *    PROGRAMM ZUR GRAFISCHEN   *
40 REM *        DARSTELLUNG VON       *
50 REM *                              *
60 REM *M A N D E L B R O T M E N G E N*
70 REM *                              *
80 REM *  HEIMO PONNATH  HAMBURG 1985 *
90 REM *                              *
100 REM********************************
110 COLOR0,1:COLOR4,1:COLOR5,6:GRAPHIC3,1:GRAPHIC5,1
120 PRINT:PRINT:PRINT:PRINTCHR$(18)"MANDELBROTMENGEN ALS GRAFIK"CHR$(146)
130 PRINT:PRINT:PRINT"PARAMETER DER GAUSSEBENE:":INPUT"RU,RO,IU,IO=";RU,RO,IU,IO
135 PRINT:PRINT"KOMPLEXE VARIABLE (NORMAL 0,0)":INPUT"X,Y=";XR,YI
140 PRINT:PRINT:PRINT"MAXIMALE ITERATIONEN:":INPUT"NMAX=";NM%
150 PRINT:PRINT:PRINT"FARBEN NACH STEIGENDEM N:":INPUT"F1,F2,F3,F4=";F1%,F2%,F3%
,F4%
160 PRINT:PRINT:PRINT"ES WIRD JETZT EINE GANZE WEILE DAUERN!!"
170 SLEEP5:GRAPHIC3:FAST:COLOR0,F4%:COLOR4,F4%:COLOR1,F3%:COLOR2,F2%:COLOR3,F1%
180 DA=(RO-RU)/159:DB=(IO-IU)/199
190 B=IU-DB
200 FORU=0TO199
210 B=B+DB:A=RU-DA
220 FORV=0TO159
230 A=A+DA
240 N%=0:R=XR:I=YI:D=0
250 DO WHILE D<4
260 X=R:Y=I
270 R=X*X-Y*Y+A:I=2*X*Y+B:D=R*R+I*I
280 N%=N%+1
290 IFN%=NM%THENEXIT
300 LOOP
310 IFN%=NM%THENDRAW0,V,U:ELSEDRAW(FNMD(N%))+1,V,U
330 NEXTV:NEXTU
340 SLOW
390 GETKEYA$:IFA$<>"←"THEN390
400 GRAPHIC5:PRINT:PRINT"BILD ABSPEICHERN?(J/N)"
410 GETKEYA$:IFA$="D"THENDIRECTORY:GOTO400
415 IFA$="J"THENBEGIN
420 INPUT"BILDNAME";B$
430 BSAVE(B$),ONB0,P7168TOP16383
440 BEND
450 END
```

3.4 Fractal-Praxis

Das Zeichnen von Fractals ist eine Geduldsprobe: Je nach Anzahl der Iterationen und ausgesuchtem Gauß-Ebenen-Abschnitt müssen Sie mit der Blockade Ihres Computers für zwei bis acht Stunden rechnen! Was wäre da zu tun? Die einzige Lösung ist das Programmieren in Maschinensprache. Weil wir zur Berechnung allerlei Operationen mit Fließkommazahlen brauchen, ist das sinnvoll nur unter Verwendung von Interpreter-Routinen zu programmieren. Die Schöpfer der Commodore-128-Firmware sind leider sehr zurückhaltend mit Informationen, so daß ein Programm, das auf diese Interpreter-Routinen zugreift, noch etwas auf sich warten lassen wird. Für den C64-Modus hat G. Pehland in der Zeitschrift "64er" (Ausgabe 11/1985, S.80) ein schönes Programm veröffentlicht. Besitzer des Commodore 128 können dieses Programm noch erheblich beschleunigen, indem sie im 64er-Modus auf doppelte Geschwindigkeit schalten. Das ist möglich durch folgende POKEs:

POKE 53296,1	schnell
POKE 53296,0	normal

Im Programm von G. Pehland müßten diese POKEs wie folgt angeordnet werden:

Zeile 1141 vor dem Befehl SYS CL umschalten auf doppelte Geschwindigkeit und Zeile 1150 nach dem Befehl SYS BG zurückschalten auf normale Geschwindigkeit. Trotz dieser Beschleunigungsmaßnahmen ist auch hier noch das Zeichnen von Fractals eine arge Geduldsprobe.

Pehlands Programm verwendet übrigens einen etwas anderen Algorithmus zum Fractal-Zeichnen, weshalb sein Apfelmännchen umgedreht auf dem Bildschirm erscheint. Eine feine Sache - die Sie auch leicht in unser Programm MANDELBROT2 einbauen können - ist die Möglichkeit, Ausschnitte eines fertigen Bildes festzulegen für eine weiteres Bild. Pehland verwendet dazu zwei Sprites, die mittels der Cursortasten an die linke obere und die rechte untere Ecke des gewünschten Ausschnittes gesteuert werden. Nach Tastendruck rechnet der Computer die Spriteposition um in Koordinaten der Gauß-Ebene.

Ein komplettes "Apfelmännchen" (Bild 3.2) dient hauptsächlich zur Übersicht. Sie erhalten es durch die Eingaben

RU = -2
RO = 0.5
IU = -1.25
IO = 1.25

Viel interessanter aber ist der Grenzbereich der (schwarz gezeichneten) Mandelbrot-Menge. Je stärker die Vergrößerung des Ausschnittes wird, desto mehr Einzelheiten sind auf dem Bild zu finden. Die Anzahl der Iterationen drückt sich in der Feinheit der Details aus. Je höher sie gewählt wird, desto mehr Einzelheiten zeigt das Bild. Allerdings sind uns durch die karge Auflösung (160 mal 200 Bildpunkte) schnell Grenzen gesetzt. Wählen wir eine zu hohe Anzahl von Iterationen, dann wird das Bild überladen, und Informationen werden zugedeckt. Bild 3.4 zeigt uns ein Beispiel dafür (Siehe Farbteil II, Fractal in zwei Auflösungen)

Im oberen Bildteil ist die Iterationszahl mit 300, im unteren mit 100 gewählt worden. Weniger Iterationen zeigen also oft mehr Information. Die optimale Schrittzahl zu finden, ist eine Sache des Ausprobierens.

"Schiefe" Mandelbrot-Mengen ergeben sich, wenn wir uns überlegen, welchen speziellen Algorithmus wir zur Erzeugung gewählt haben und diesen dann etwas verallgemeinern. Im Gegensatz zu MANDELBROT1 haben wir in MANDELBROT2 stillschweigend X und Y (also den Realteil und den Imaginärteil der Variablen in unserer Ausgangsgleichung m=(x+yi)^2 + (a+bi) auf den Wert 0 festgelegt. Wir haben aber schon in MANDELBROT1 erkennen können, daß sich die Verhältnisse völlig verändern, wenn X und Y andere Werte annehmen. Bauen wir in MANDELBROT2 noch ein:

```
135 INPUT"X,Y=";XR,YI
```

und ändern Zeile 240 um in

```
240 N%=0:R=XR:I=YI:D=0
```

dann schaffen wir uns neue Einflußmöglichkeiten. Die "Apfelmännchen", die wir damit erzeugen, sehen verzerrt aus oder sind häufig gar nicht mehr zu erkennen (Bild 3.5 - Siehe Farbteil III, Ein leicht verzerrtes "Apfelmännchen"). Es wurden folgende Parameter verwendet:

RU=-2;RO=0.5;IU=-1.25;IO=1.25
X=0.1;Y=0.5
Iterationszahl 100

Bild 3.6 zeigt (Siehe Farbteil IV) ein stark verzerrtes "Apelmännchen". Die Parameter dafür lauten:

RU=-.57;RO=0.7;IU=-1.5;IO=1
X=-0.6;Y=0.7
Iterationszahl 100

Die Erforschung der Randgebiete dieser verzerrten Mandelbrot-Mengen dürfte allerlei interessante neue Bilder ergeben. Viel Spaß beim Ausprobieren neuer Kombinationen, von denen es unendlich viele gibt, und einen widerstandsfähigen Computer, der den Rund-um-die-Uhr-Betrieb nicht übelnimmt, wünsche ich Ihnen!

3.5 Literatur zu Kapitel 3

(1) G. Pehland: Bilder aus einer anderen Dimension; 64er, Ausg.11, 1985, S.80

(2) A. K. Dewdney: Computer-Kurzweil; Spektrum der Wissenschaft, Ausg.10, 1985, S.8

(3) H.-O. Peitgen, P.H. Richter: The Beauty of Fractals, Berlin/Heidelberg/New York/Tokyo 1985, Springer-Verlag (Vorabinformation)

(4) F. Fricker: Barnard-Medaille für Benoit Mandelbrot; Spektrum der Wissenschaft, Ausg.11, 1985, S.14

(5) F. Capra: Wendezeit; Bern/München/Wien 1985, Scherz-Verlag

(6) K. Klotz: Computergrafik zum Nachmachen; CHIP, Ausg.10, (1984), S.38

(7) M. L. Prueitt: Art and the Computer; New York 1984, Mc-Graw-Hill

(8) J. D. Foley, A. van Dam: Fundamentals of interactive Computer Graphics; Reading.Mass. 1984, Addison-Wesley

(9) B. B. Mandelbrot: Fractals Form, Chance, and Dimension; San Francisco 1977, Freeman

(10) B. B. Mandelbrot: The Fractal Geometry of Nature; San Francisco, Freeman

4 Der Tastaturpuffer und seine Anwendungen

Es gibt manches Mal im Dasein von Computern Situationen, in denen sie Eingaben per Tastatur nicht sofort bearbeiten können; während eines Prokgrammablaufes beispielsweise oder bei einem Ladevorgang. Lediglich einige wenige Tasten sind immer aktiv, die STOP-Taste zum Beispiel. Trotzdem lohnt sich der Griff in die Tasten, denn es wird nichts vergessen. Die fraglichen Zeichen sind nur beiseite gelegt, bis sie bearbeitet werden können. Diese Ablage, in der sie warten, nennt man den Tastaturpuffer. Ist das beendet, was unseren C128 davon abhielt, uns zuzuhören, dann wendet er sich diesem Pufferinhalt zu und arbeitet ihn durch.

Der Puffer befindet sich im Speicherbereich von 842 bis 851 (also $034A bis 0353). Außerdem braucht unser Computer eine Speicherstelle, wo ihm gezeigt wird, wie viele Zeichen er im Puffer findet. Diesen Job hat die Zelle 208 (das ist $D0). Was können wir nun mit dieser Erkenntnis anfangen? Sie werden überrascht sein, welche Möglichkeiten sich uns da auftun!

Lassen Sie uns zunächst einmal sehen, wie wir diesen Puffer benutzen können. Zunächst schaffen wir den Zustand, daß der C128 unsere Tastatureingaben nicht beachtet: Wir beschäftigen ihn mit einem Programm. Listigerweise wird das Programm etwas in seinen Tastaturpuffer schreiben:

```
10SCNCLR:PRINT:PRINT
20POKE842,ASC("?")
30POKE843,34
40POKE844,ASC("H")
50POKE845,ASC("I")
60POKE846,ASC("!")
70POKE847,34
80POKE848,13
90POKE208,7
100END
```

Geben Sie doch mal RUN ((RETURN)) ein, und sehen Sie, was geschieht: Nach einem READY taucht die Zeile ?"HI!" auf dem Bildschirm auf und darunter das Wort HI!. Weshalb? Weil wir das in den Tastaturpuffer gepoked haben. (34 ist der ASCII-Code für das Anführungszeichen und 13 der Code für die RETURN-Taste). Der Computer hat unsere Eingaben per Programm so verstanden, als hätten wir sie ihm im Direktmodus vorgelegt. Wir haben somit die Möglichkeit, einen scheinbaren Widerspruch zwanglos zu umgehen, nämlich den programmierten Direktmodus!

Nun müssen wir uns nur noch Gedanken darüber machen, was es im Direktmodus Interessantes gibt, das wir immer schon gerne per Programm erledigt hätten. Ihnen fällt bestimmt viel ein: Programmzeilen in ein bestehendes Programm einzufügen, den Monitor zu benutzen und so weiter.

Übrigens sind wir nicht unbedingt auf die 10 Speicherstellen des Tastaturpuffers festgelegt. Weitere 10, die sich daran anschließen, können im allgemeinen ebensogut mitbenutzt werden. Sie gehören zur Tabelle der Tabulator-Stops, die aber - soweit ich das bisher wahrnehmen konnte - lediglich für die Tabulatorsprungfunktion mittels CTRL-I oder TAB verwendet wird. Wenn man aber noch mehr Speicherplätze benutzt, wird es kritisch, denn da liegen einige wichtige Notizen des Computers und ab 896 könnte sogar der Lebensnerv, nämlich die CHRGET-Routine zerstört werden. Also bescheiden wir uns lieber mit 20 Plätzen im Tastaturpuffer.

Im Folgenden werden wir uns einige Anwendungen des programmierten Direktmodus ansehen (manchmal spricht man im englischen Sprachraum auch von "dynamic keyboard", also von der "kraftvollen Tastatur"). Die Programmbeispiele sollen als Module aufgebaut sein, die wir dann mittels der in Kapitel 5 vorgestellten MERGE-Funktion jederzeit in bestehende Programme einbauen können. Zuvor aber soll der Begriff des Programm-Moduls noch etwas erklärt werden.

4.1 Programm-Module

Es gibt - besonders als Ergebnis von Bemühungen der BASIC-Programmierer - Programme, die wie ein lebender Organismus gewachsen sind. Da ist es - bei längeren Schöpfungen - mitunter schwierig zu ergründen, wie und warum sie überhaupt funktionieren; und ebensowenig, wie ein Bein oder ein Arm alleine sinnvoll sind, sind es dann Programmteile. Andere Programmsprachen fördern diese doch auch irgendwie sympathische Art der Programmlabyrinthe nicht, da gehts oft streng der Reihe nach. Immer wieder gibt es auch mehr oder weniger erfolgreiche Versuche, BASIC-Programme zu strukturieren, und auch unser Computer beinhaltet einige Befehle, die dazu beitragen sollen. Ein weiterer Schritt in diese Richtung ist der Aufbau von Programmen aus Bausteinen, den Programm-Modulen.

Die Zielvorstellung wäre ein Programm, aus lauter solchen Modulen zusammengesetzt, die hätte der erfahrene Programmierer alle schon fertig in einer Modulbibliothek auf Diskette vorliegen, die nur noch durch einen Rahmen zusammenzubinden wären. Wie müßte solch ein Modul aussehen und vor allem, welche Angaben müßte die begleitende Dokumentation enthalten?

Das Modul: Es sollte möglichst allgemein gehalten und daher vielseitig verwendbar sein. Nötige Anpassungen sollten leicht durchführbar sein, weshalb auch der Aufbau des Moduls überschaubar zu halten ist.

Die Dokumentation: Nach mehr oder weniger langer Zeit hat jeder Programmierer vergessen, was er da geschrieben hat. In der Dokumentation muß daher enthalten sein:

1. Was leistet das Modul?

2. Welche Variablen werden wie verwendet?

   a. Variable, die ins Modul hineingegeben werden und
   b. Variable, die aus dem Modul herausgegeben werden.
   c. Variable, die im Modul erzeugt werden und entweder globale oder nur lokale Verwendung finden.

3. Verwendung des Moduls:

   a. Wie wird es in das Bindeprogramm eingefügt und worauf ist dabei eventuell zu achten?
   b. Wie kann das Modul vom Hauptprogramm her in Betrieb genommen werden?

Es gibt sicher noch weitere Punkte, die manchmal Bedeutung haben: eine spezielle Angabe von Fehlern, die im Modul auftreten können beispielsweise oder einen Hinweis auf andere Module etc.

Nach all diesen Vorbemerkungen sehen wir uns nun einige Module an, die den programmierten Direktmodus verwenden.

4.1.1 MODUL: ZEILEN EINFÜGEN

1. Was leistet das Modul?

   In ein bestehendes Programm werden durch programmierten Direktmodus zwei neue Zeilen eingefügt, die eine zuvor eingegebene Funktion definieren.

2. Verwendung von Variablen:

   Die Variable Z1 muß in das Modul eingeführt werden. Z1 ist die Zeilennummer, in der wir den Funktionsstring abzulegen wünschen. Z1+10 enthält die Funktionsdefinition, und Z1-10 ist die Zeilennummer, mit der der Neustart (siehe bei Einbindung des Moduls) des Programm erfolgt.

   Zwei weitere Variable und eine Funktion werden im Modul definiert:

F$	=	String, welcher die Funktion enthält.
FN F(X)	=	Funktion, in welcher F$ verwendet wird.
X	=	Variable der Funktion.

   Alle Variablen (und die Funktion) haben globale Bedeutung.

3. Einbindung des Moduls und Anwendung:

   Die Einbindung ist an jeder beliebigen Stelle des Programm möglich, die nicht in einem Unterprogramm oder einer Schleife steht. Zuvor sollte noch ein Z1 definiert sein, das größer als 10 ist und sicherstellt, daß Z1 und Z1+10 freie Zeilennummern sind.

   Das Modul kann sowohl im direkten Programmablauf als auch durch GOTO angesteuert werden. Dies ist die Funktionsweise:

- Der Bildschirm wird gelöscht und eine Funktion Y=F(X) abgefragt.
- Nach erneutem Löschen des Bildschirms und Angleichen der Zeichenfarbe an den Hintergrund (hier als schwarz angenommen) wird in der 3. Bildschirmzeile gedruckt:

  (Zeilennummer Z1) F$="eingeg. Funktion"

  In der 4.Zeile:

  (Zeilennummer Z1+10) DEF FN F(X)=-"-

  In Zeile 5 schließlich:

  RUN (Zeilennummer Z1 - 10)

Der Cursor wandert in die Home-Position, 3 RETURNs (das ist CHR$(13)) in den Tastaturpuffer und in die Speicherstelle 208 diese Anzahl von drei.

Das Programm endet nun, und auf dem Bildschirm erscheint (ebenfalls unsichtbar) READY. Der Cursor steht nun auf Zeile 3. Der Tastaturpuffer wird abgearbeitet, was bedeutet, daß die Inhalte der Bildschirmzeilen 3 und 4 durch die zwei RETURNs übernommen und das RUN-Kommando ausgeführt wird.

Dieser Neustart löscht den Bildschirm und setzt die Zeichenfarbe auf einen sichtbaren Wert (hier auf Weiß).

Nach dieser ausführlichen Funktionsbeschreibung sollen Sie nun auch das Modul eintippen können. Als "ZEILEN EINF MOD" finden Sie es hier noch zum Ausprobieren mit einer Zeile 1, die der Variablen Z1 den Wert 110 zuordnet:

Programm ZEILEN EINF MOD

Ein Programm-Modul für selbstmodifizierende Programme

```
1 Z1=110
10 REM ***** MODUL ZEILEN EINFUEGEN *****
20 PRINTCHR$(147)CHR$(17)CHR$(17)
30 PRINT"WELCHE FUNKTION ?"
40 INPUT"Y=F(X)=";F$
50 PRINTCHR$(147)CHR$(17)CHR$(144)
60 PRINTZ1"F$="CHR$(34)F$CHR$(34)
70 PRINTZ1+10"DEF FN F(X)="F$
80 PRINT"RUN"Z1-10CHR$(19);
90 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE208,3:END
100 PRINTCHR$(147)CHR$(5):LIST
110 F$="X*X"
120 DEF FN F(X)=X*X
```

Mit Hilfe dieses und ähnlicher Module ist es möglich, selbstmodifizierende Programme zu realisieren. Bedenken Sie, daß wir damit bei jedem Durchlauf bis zu 20 neue Programmzeilen übernehmen könnten, daß wir in "laufenden" Programmen ganze Abschnitte umzuschreiben imstande sind,...

4.1.2 Modul: Monitoraufruf

Normalerweise ist ein BASIC-Programmablauf in dem Moment beendet, in dem der MONITOR-Befehl bearbeitet ist. Dann meldet sich der Monitor mit einer Registeranzeige, und wir befinden uns im Direktmodus. Weil wir diesen aber nun per Programm beherrschen, können wir jetzt auch Funktionen des Monitors, wie in diesem Beispiel den Hexdump von Speicherteilen, in Programme einbinden.

1. Was leistet das Modul?

   Es erlaubt die Verwendung des Monitor-Kommandos M zur Anzeige von Speicherinhalten.

2. Variable:

   Lediglich zwei Stringvariable spielen eine Rolle. Sie werden vor dem Modulaufruf definiert und bezeichnen die erste anzuzeigende Speicherstelle in Hexadezimalform.

   Dabei ist Z0$ die im Monitor verwendete Bank-Kennziffer, also die vordere Stelle der Hex-Adresse. Beispielsweise ist bei $FD800 für Z0$ zu setzen "F".

   Z1$ ist die vierstellige Hexzahl, die sich an Z0$ anschließt. Im obigen Beispiel also Z1$="D800".

3. Einbindung und Verwendung des Moduls:

   Ebenso wie das vorhin vorgestellte Modul ist auch dieses an jede beliebige Stelle des Programm zu plazieren, außer in Unterprogramme oder Schleifen.

   Es kann direkt im Programmlauf oder durch GOTO aktiviert werden.

   Hier nun die Erklärung des Ablaufs, die aber nicht ganz so ausführlich wie beim ersten Beispiel sein wird:

Nach dem Löschen des Bildschirms und dem Überspringen der READY-Zeile werden nacheinander das Monitorkommando, der Monitorbefehl M, der Befehl zum Verlassen des Monitors X und ein RUN100 auf den Bildschirm gedruckt. Eine bestimmte Anzahl von Leerzeilen ist hier nötig, um die Registeranzeige und die zu druckenden Zeilen der Speicherinhalte zu überspringen. Je nach verwendetem Bildschirm sind das letztere dann 4 Zeilen (80-Zeichen, hier werden 16 Kolonnen ausgegeben) oder 7 Zeilen (40-Zeichen, wobei wir nur 8 Kolonnen erhalten). Welcher Bildschirm aktiv ist, kann der Computer selbst herausfinden, indem er sich den Inhalt der Speicherstelle 238 ($EE) ansieht. Dort findet er die höchste Spaltenziffer: 79 beim 80-Zeichen-Betrieb und 39 im 40-Zeichen-Betrieb. In Programmzeile 50 fügt er im Bedarfsfall noch die nötige Anzahl von Cursor-Down-Kommandos hinzu.

In Zeile 40 wird das M-Kommando gedruckt. Hier berechnet der Computer noch die zweite M-Adresse, die um $30 höher als die Startadresse gewählt wurde, um den gesamten Ausdruck auf den 40-Zeichen-Bildschirm bringen zu können. Falls Sie für eigene Anwendungen einmal einen anderen Wert als $30 benötigen, müssen Sie unter Umständen noch die Anzahl der CHR$ (17)-Kommandos verändern.

In Zeile 70 kann anstelle der Zahl 100 auch eine Variable eingefügt werden, die vor dem Modulaufruf zu definieren ist. Dadurch kann die weitere Verarbeitung nach dem Modulablauf noch flexibler gestaltet sein. Im hier abgedruckten Programm "MONITOR MOD"

Programm MONITOR MOD

Programm-Modul zur Verwendung der Monitorfunkton M in BASIC-Programmen

```
1 Z1$="1C00":Z0$="0"
10 REM ***** PROGR.DIREKTMODUS : MONITORAUFRUF *****
20 PRINTCHR$(147)CHR$(17)
30 PRINT"MONITOR"CHR$(17)CHR$(17)CHR$(17)CHR$(17)
40 PRINT"M ";Z0$+Z1$;" ";Z0$+HEX$(DEC(Z1$)+DEC("30"))CHR$(17)CHR$(17)CHR$(17)CHR
$(17)
50 BANK0:IFPEEK(238)=39THENPRINTCHR$(17)CHR$(17)
60 PRINT"X"CHR$(17)
70 PRINT"RUN100"
80 PRINTCHR$(19);
90 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE845,13:POKE208,4:END
95 REM *********************************************
100 LIST10
110 PRINTCHR$(17)"DAS WARS!"
120 END
```

sind - um es als Beispiel lauffähig zu machen - noch 4 Zeilen hinzugefügt worden: Zeile 1, welche die Stringvariablen definiert, und die Zeilen 100 bis 120. Die Voreinstellungen zeigen dann im Programmablauf den Anfang des Programm als Hex-Listing.

4.1.3 Modul: Transferbefehl

Im Monitor existiert ein besonders starker Befehl, das T-Kommando. Damit können beliebig große Speicherbereiche verschoben werden. Zwar gibt es auch in BASIC 7.0 drei Befehle, die das können, nämlich STASH, FETCH und SWAP. Leider aber sind diese Befehle nicht geeignet, Verschiebungen innerhalb der Banks 0 oder 1 oder der beiden untereinander vorzunehmen. Sie beziehen sich auf höhere Banks, die erst mit den Speichererweiterungen erreichbar sind.

Damit hat es nun ein Ende, und falls Sie einmal beispielsweise eine Bit-Map aus Bank 0 nach Bank 1 verschieben möchten, können Sie das mit diesem Modul erledigen.

1. Was leistet das Modul?

   Beliebige Speicherbereiche, werden an beliebige Zieladressen kopiert.

2. Variable:

   Insgesamt spielen 6 Variable eine Rolle, die vor dem Modul-Aufruf definiert sein müssen:

   Z0$ bis Z4$ sind Stringvariable, die die Adressen für den Transferbefehl enthalten in Hexadezimalzahlen. Folgende Zuordnung ergibt sich aus dem Transferbeispiel:

   T 02000 03030 12F00

   Das bedeutet, daß der Speicherbereich zwischen $02000 und $03030 nach oben verschoben (in Wirklichkeit: kopiert) wird, und zwar aus der Bank 0 in die Bank 1 - ab $12F00 und folgende.

Z0$	=	"0"	Bank, aus der verschoben wird
Z1$	=	"2000"	Quelle Startadresse
Z2$	=	"3030"	Quelle Endadresse
Z3$	=	"1"	Bank, in die hinein verschoben wird
Z4$	=	"2F00"	Ziel Startadresse

   Eine weitere Variable ist Z:

   Z ist die Zeilennummer, von der an das Programm sinnvollerweise neu gestartet wird.

3. Einbindung und Verwendung:

   Das Modul kann an beliebiger Stelle eines Programm eingesetzt werden, nur nicht in Unterprogrammen oder Schleifen.

Im abgedruckten Listing "TRANSFER MOD" sind vor das Modul zur Demonstration noch einige Grafikbefehle und die Definition der Variablen gehängt (Zeilen 1 bis 6). Ab Zeile 90 beginnt wieder das Hauptprogramm.

Programm TRANSFER MOD

Ein Modul zum Verschieben von Speicherbereichen

```
1 REM ***** PROG. DIREKTMODUS : TRANSFERBEFEHL *****
2 COLOR0,1:COLOR1,3:COLOR4,1:GRAPHIC1,1
3 FORI=1TO30
4 DRAW1,I,0TO10*I,100
5 NEXTI:COLOR1,6:WIDTH2:CIRCLE1,230,50,20,20
6 Z0$="0":Z1$="2000":Z2$="3030":Z3$="0":Z4$="2F00":Z=100
10 REM ******** TRANSFER MODUL ********
20 PRINTCHR$(147)CHR$(17)
30 PRINT"MONITOR"CHR$(17)CHR$(17)CHR$(17)CHR$(17)
40 PRINT"T ";Z0$+Z1$;" ";Z0$+Z2$;" ";Z3$+Z4$;CHR$(17)
50 PRINT"X"CHR$(17)
60 PRINT"RUN "Z
70 PRINTCHR$(19);
80 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE845,13:POKE208,4:END
90 REM ***** WEITER MIT BASIC *****
100 LIST10
110 PRINTCHR$(17)"DAS WARS!"
120 END
```

Die Demonstration zeichnet einige Dinge auf die obere Hälfte des Grafik-Bildschirmes und kopiert sie dann durch das Transfer-Modul in die untere Hälfte. Auf diese Weise ist auch zu erkennen, daß es sich hier um ein KOPIEREN, nicht um ein wirkliches VERSCHIEBEN handelt, denn das Bild auf der oberen Hälfte bleibt ja erhalten.

Das Demonstrationsprogramm ist für den Betrieb mit 2 Bildschirmen geschrieben. Sollten Sie lediglich mit dem 40-Zeichen-Monitor arbeiten, dann sollten Sie in Zeile 95 noch den Befehl GRAPHIC 0 einfügen.

Ein Problem gibt es noch: Nach jedem RUN sind bekanntlich immer alle Variablen eines Programm gelöscht. Häufig stört das nicht weiter, weil ohnehin die Modifikation zu Beginn eines Programmablaufes eingebaut wird oder man durch ein GOSUB in die Zeile mit den Variablendefinitionen schnell wieder voreingestellte Variable zurückholen kann. Manchmal - besonders, wenn man ohne an die besondere Eigenart der Module zu denken diese irgendwo in einem Programm verwendet - kann es aber schon zur bösen Überraschung werden, plötzlich bar aller Variablen dazustehen. Nichts hindert uns dann aber, statt durch RUN das Programm durch GOTO neu anlaufen zu lassen. Ersetzen Sie in solchen Fällen also einfach die Zeile

70 PRINT"RUN "Z (oder ähnliche)

durch

70 PRINT"GOTO"Z

Die Variablen sind dann alle noch präsent, ja man kann nun auch die Module innerhalb von Schleifen aufrufen!

4.2 2D-FUNKTIONEN: Ein Programm mit Modulen

Als Beispiel für ein Programm mit solchen Modulen ist nachstehend das Listing "2D-FUNKTIONEN" abgedruckt:

Programm 2D-FUNKTIONEN

Programm zur grafischen Darstellung von zweidimensionalen Funktionen

```
10 CLR:TRAP650
20 REM ***************************************************************
30 REM *                                                             *
40 REM *        GRAFISCHE DARSTELLUNG BELIEBIGER 2D-FUNKTIONEN       *
50 REM *                                                             *
60 REM *                HEIMO PONNATH  HAMBURG  1985                 *
70 REM *                                                             *
80 REM ***************************************************************
90 COLOR0,1:COLOR1,8:COLOR4,1:COLOR5,2:COLOR6,1:Z1=250
95 GRAPHIC1,1:GRAPHIC5,1
100 BANK15:SYS65520,,10,15:PRINT"GRAFISCHE DARSTELLUNG BELIEBIGER 2D-FUNKTIONEN"
110 BANK15:SYS65520,,15,10:PRINT"DIESE FUNKTION IST PROGRAMMIERT:":K=1:GOSUB250
120 PRINT:PRINT"Y = F(X) ="F$:PRINT:PRINT,"SOLLS EINE ANDERE SEIN (J/N) ?";
130 GETA$:IFA$<>"J"ANDA$<>"N"THEN130
140 IFA$="N"THEN240
150 FORI=1TO18:PRINTCHR$(27)+"Y":NEXTI:BANK15:SYS65520,,10,10
160 REM ***** MODUL ZEILEN EINFUEGEN *****
170 PRINT"WELCHE FUNKTION ?":PRINT
180 INPUT"Y = F(X) =";F$:FAST
190 PRINTCHR$(147)CHR$(17)CHR$(144)
```

```
200 PRINTZ1"F$="CHR$(34)F$CHR$(34)
210 PRINTZ1+10"DEF FN F(X)="F$
220 PRINT"RUN"Z1-10CHR$(13);
230 BANK0:POKE842,13:POKE843,13:POKE844,13:POKE208,3:END
240 PRINTCHR$(147)CHR$(5):SLOW:K=0
250 F$="EXP(COS(1/X))"
260 DEF FN F(X)=EXP(COS(1/X))
270 IFK=1THEN RETURN
280 REM ***** MODUL TRANSFORMATION *****
290 PRINTCHR$(17)CHR$(17)"SYSTEMGRENZWERTE:"CHR$(17)
300 INPUT"XU,XO,YU,YO=";XU,XO,YU,YO
310 SX=319/(XO-XU):SY=-199/(YO-YU):TX=-XU*SX:TY=-YO*SY
320 DEF FN TX(X)=SX*X+TX
330 DEF FN TY(Y)=SY*Y+TY
340 REM ***** ZEICHNEN DES KOORDINATENSYSTEMS *****
350 GRAPHIC1:COLOR1,12
360 IF (XO-XU)>30 THEN 420:ELSE BEGIN
370 :FOR X=INT(XU) TO INT(XO)
380 ::IFX=0THEN400
390 ::DRAW1,FN TX(X),FN TY(YU) TO FN TX(X),FN TY(YO)
400 :NEXTX
410 BEND
420 IF (YO-YU)>30 THEN 480:ELSE BEGIN
430 :FOR Y=INT(YU) TO INT(YO)
440 ::IFY=0THEN460
450 ::DRAW1,FN TX(XU),FN TY(Y) TO FN TX(XO),FN TY(Y)
460 :NEXTY
470 BEND
480 COLOR1,3:WIDTH2
490 DRAW1,FN TX(XU),FN TY(0) TO FN TX(XO),FN TY(0)
500 DRAW1,FN TX(0),FN TY(YU) TO FN TX(0),FN TY(YO)
510 WIDTH1
520 REM ***** ZEICHNEN DER FUNKTION *****
530 COLOR1,6:LOCATE FN TX(XU),FN TY(FN F(XU))
540 FOR X=XU TO XO STEP 1/SX
550 :Y=FNF(X)
560 :IF FN TY(Y)<0 OR FN TY(Y)>199 THEN 610
570 :DRAW1 TO FN TX(X),FN TY(Y)
580 NEXTX
590 CHAR1,0,0,"Y="+F$,1
600 END
610 IF FN TY(Y)>199 THEN LOCATE FN TX(X+1/SX),FN TY(YU):ELSE BEGIN
620 :LOCATE FN TX(X+1/SX),FN TY(YO)
630 BEND
640 GOTO580
650 REM ***** FEHLERBEHANDLUNG *****
660 IF ER=14 THEN RESUME 580

READY.
```

Es erlaubt Ihnen innerhalb gewisser Grenzen, damit das Listing nicht zu umfangreich wird, die grafische Darstellung beliebiger zweidimensionaler Funktionen. Ein weiteres Modul wird hier angewendet, das Modul TRANSFORM MOD:

Programm TRANSFORM MOD

Programm-Modul zur Transformation beliebiger Koordinatensysteme in das Bildschirmsystem

```
10 REM ***** MODUL TRANSFORMATION *****
20 PRINTCHR$(147)CHR$(17)"SYSTEMGRENZWERTE:"
30 INPUT"XU,XO,YU,YO=";XU,XO,YU,YO
40 SX=319/(XO-XU):SY=-199/(YO-YU):TX=-XU*SX:TY=-YO*SY
50 DEF FN TX(X)=SX*X+TX
60 DEF FN TY(Y)=SY*Y+TY

READY.
```

Hier ist seine Beschreibung:

1. Was leistet das Modul?

   Es erfragt vom Benutzer die Grenzwerte eines Koordinatensystems und erzeugt zwei Funktionen, die die Transformation beliebiger Punkte des angegebenen Systems in Bildschirmkoordinaten vornehmen können.

2. Variable:

   Alle benötigten Variablen werden im Modul erzeugt:

   XU,XO kleinste und größte X-Koordinate

   YU,YO dasselbe für die Y-Koordinaten des gewünschten Systems

   SX,SY Interne Variable. Das sind die Skalierungsfaktoren in X- und in Y-Richtung.

   TX,TY ebenfalls interne Variable. Hier dreht es sich um die Translation in X- und in Y-Richtung.

   Um diese vier internen Variablen braucht man sich normalerweise nicht zu kümmern, Sie werden automatisch erzeugt und verwendet durch die beiden Funktionen:

   FN TX(X) Transformiert eingegebene Koordinaten des gewählten Systems (X) in Bildschirm-Koordinaten um,

   FN TY(Y) leistet dasselbe für die Y-Richtung.

3. Einbau und Verwendung des Moduls:

   Das Modul ist an beliebiger Stelle in Programme einzusetzen, muß aber vor der Verwendung der Funktionen im Programmlauf angesteuert werden, weil sonst ein UNDEFN'D FUNCTION ERROR auftritt.

   Es kann auf beliebige Weise verwendet werden.

Eine Erklärung der mathematischen Grundlagen an dieser Stelle wäre etwas umfangreich. Falls Sie daran interessiert sind, lesen Sie bitte in der Serie "Grafik-Streifzüge" in der Zeitschrift 64er nach, wo alles Wissenswerte über Transformationen auf einfache Weise erklärt wird. Sie lernen dort auch, wie sich mit einfachen Mitteln Rotationen ins Modul einbauen lassen.

Sehen wir uns nun nochmal das Programm "2D-FUNKTIONEN" an. Es ist gewissermaßen die Sparausführung eines solchen Grafikprogrammes. So kann man beispielsweise nicht bestimmte Bereiche beim Zeichnen ausklammern. Es wird immer von XU bis XO gezeichnet. Auch ist es nicht möglich, sowohl XU als auch XO als positive Werte einzugeben, ebensowenig, wie es möglich ist, YU und YO beide negativ anzugeben. Jedesmal müssen der untere und der obere Wert verschiedene Vorzeichen haben (aber auch noch die Null als höchster oder niedrigster Wert wird akzeptiert). Das Programm wurde für den Betrieb mit 2 Bildschirmen geschrieben. Sollten Sie lediglich den 40-Zeichen-Schirm verwenden, müssen Sie nur in Zeile 95 statt GRAPHIC5,1 nun GRAPHIC0,1 schreiben und in Zeile 600 eine Zurückschaltung in den Grafik-Modus 0 veranlassen. Beim Zeichnen werden Sie bemerken, daß sich ein Teil des Koordinaten-Rasters verfärbt. Das liegt daran, daß unser Programm mehrere Farben verwendet und nicht im Multicolormodus läuft, um keine Einbußen bei der Auflösung hinnehmen zu müssen. Weil aber die Farbgebung immer in 8*8 Bit-Feldern geschieht, wird eine alte Zeichnung immer dann neu gefärbt, wenn eine neue Linie in anderer Farbe durch dieses Feld läuft. Viel Spaß wünsche ich Ihnen beim Ausbauen dieser Sparversion zum professionellen 2D-Funktionen-Programm.

5 Nützliches Sammelsurium

5.1 Simulieren eines PRINT AT

Der neue SYS-Befehl im 128er-Modus erlaubt die Übergabe der Registerinhalte an aufgerufene Routinen:

```
SYS adresse,A,X,Y,S
```

Dabei ist

A	= Akkumulator
X	= X-Register
Y	= Y-Register
S	= Prozessorstatusregister

An der Adresse 65520 befindet sich eine Routine, die den Cursor an die Stelle setzt, die durch das X-Register (Zeile) und das Y-Register (Spalte) vorgegeben wird. Deshalb kann ein PRINT AT simuliert werden durch die Befehlssequenz

```
BANK15:SYS 65520,,Zeile,Spalte:PRINT A$
```

A$ wird beispielsweise in die 10.Zeile ab der 5.Spalte gedruckt, wenn man schreibt:

```
BANK15:SYS 65520,,10,5:PRINT A$
```

5.2 Welche Bank haben wir?

Sollten Sie auch zu denen gehören, die die Umschaltung zwischen den Banks des Commodore 128 interessant finden, dann wird es Ihnen vermutlich auch so ergangen sein wie mir, daß Sie nämlich irgendwann nicht mehr so genau wissen, in welcher Bank Sie nun gerade operieren, und erstaunt sind, daß beispielsweise ein POKE nicht das erwünschte Ergebnis bringt, weil die falsche Bank eingeschaltet war. Da stolperte ich über eine Adresse in der erweiterten Zeropage, in der offenbar immer die gerade aktuelle Bank gespeichert ist. Durch

```
PRINT PEEK(981)
```

behalten Sie immer den Überblick.

5.3 Ausgabe von Fehlermeldungen

Vermutlich ist diese Methode von BASIC aus weit weniger interessant als in Assemblerprogrammen: Durch

```
BANK15:SYS 19775,,X
```

(die 2 Kommas gehören da wirklich hin!) erzwingt man die Ausgabe der Fehlermeldung mit der Nummer X.

In Assembler lädt man die Fehlernummer in das X-Register und springt dann die Ausgaberoutine bei $4D3F an:

```
LDX #$17
JMP $4D3F
```

erzwingt einen STRING TOO LONG ERROR und führt den Computer in den READY-Status. Allerdings sollte man auch hier noch auf die richtige Speicherkonfiguration achten: Die BASIC-ROMs müssen eingeschaltet sein.

5.4 Eine OLD-Routine

Versehentlich gelöschte BASIC-Programme wieder LIST- und lauffähig zu machen durch einen OLD-Befehl, das sieht unser BASIC 7.0 nicht vor. Dabei passiert es gerade bei den ersten Versuchen am PC128 so leicht, daß man in die falsche Bank rutscht und dann den Computer scheintot macht durch ein kleines POKE-Kommando. Da gibt es dann den Retter in der Not, den RESET-Knopf, dessen Betätigung aber das BASIC-Programm unauffindbar macht. Sofort habe ich mich an eine OLD-Routine gesetzt.

Auch beim PC128 wird nur der erste Zeilenlinker durch 2 Nullen überschrieben. Es gilt also, diesen zu restaurieren. Außerdem muß der Interpreter erfahren, wo das Ende des Programms zu finden ist. Hier geht das nicht mehr durch den Zeiger, der den Beginn der Variablen andeutet - die liegen jetzt in der Bank 1 -, sondern dazu existiert ein neuer Zeiger. Hier eine Zusammenstellung der für OLD wichtigen Vektoren:

$2D/2E	dez. 45/46	BASIC-Progr.-Start
$1210/1	dez. 4624/5	BASIC-Progr.-Ende

Das im folgenden abgedruckte Programm OLD sucht zuerst den Anfang der 2. Programmzeile. Sollte das Programm durch allerlei Manipulationen schon so zerstört sein, daß diese Suche erfolglos bleibt, wird ein STRING TOO LONG ERROR ausgegeben. Andernfalls restauriert OLD den ersten Zeilenlinker und durchsucht dann den Textspeicher, bis die 3 Nullen gefunden werden, die ein Programmende markieren. Die Endadresse wird nun noch in den Endevektor 1210/1 geschrieben, und unser Mißgeschick ist repariert.

Programm OLD

Ein durch NEW oder RESET versehentlich gelöschtes BAISC-Programm wird wieder einsatzfähig

```
MONITOR
    PC  SR AC XR YR SP
; QFB000 00 00 00 00 F8

Q. 01300  A5 2D      LDA $2D
Q. 01302  18         CLC
Q. 01303  69 04      ADC #$04
Q. 01305  85 24      STA $24
Q. 01307  A5 2E      LDA $2E
Q. 01309  69 00      ADC #$00
Q. 0130B  85 25      STA $25
Q. 0130D  A0 00      LDY #$00
Q. 0130F  B1 24      LDA ($24),Y
Q. 01311  F0 08      BEQ $131B
Q. 01313  C8         INY
Q. 01314  C0 A0      CPY #$A0
Q. 01316  D0 F7      BNE $130F
Q. 01318  4C 3F 4D JMP $4D3F
Q. 0131B  C8         INY
Q. 0131C  98         TYA
Q. 0131D  A0 00      LDY #$00
Q. 0131F  18         CLC
Q. 01320  65 24      ADC $24
Q. 01322  91 2D      STA ($2D),Y
Q. 01324  85 24      STA $24
Q. 01326  90 02      BCC $132A
Q. 01328  E6 25      INC $25
Q. 0132A  A5 25      LDA $25
Q. 0132C  C8         INY
Q. 0132D  91 2D      STA ($2D),Y
Q. 0132F  88         DEY
Q. 01330  B1 24      LDA ($24),Y
Q. 01332  AA         TAX
Q. 01333  C8         INY
Q. 01334  B1 24      LDA ($24),Y
Q. 01336  F0 07      BEQ $133F
Q. 01338  85 25      STA $25
Q. 0133A  86 24      STX $24
Q. 0133C  38         SEC
Q. 0133D  B0 F0      BCS $132F
Q. 0133F  A5 24      LDA $24
Q. 01341  18         CLC
Q. 01342  69 02      ADC #$02
Q. 01344  8D 10 12 STA $1210
Q. 01347  90 07      BCC $1350
Q. 01349  E6 25      INC $25
Q. 0134B  A5 25      LDA $25
Q. 0134D  8D 11 12 STA $1211
Q. 01350  60         RTS
Q. 01351  00         BRK
```

OLD ist hier genauso abgedruckt, wie Sie es mittels des Monitors eingeben können. Drücken Sie zuerst die F8-Taste (dann schaltet sich der Monitor ein), und geben Sie dann ein:

A 0E000 LDA $2D ((RETURN))

Die nächste Adresse erscheint nun automatisch, und Sie brauchen nur noch die weiteren Befehlssequenzen abzuschreiben. OLD ist willkürlich nach E000 gelegt worden. Falls Sie das Programm lieber an anderer Stelle hätten, können Sie es mit dem T-Befehl des Monitors auch schnell verschieben. Sollten Sie es beispielsweise lieber bei $D000 liegen haben, dann verwenden Sie

T E000 E051 D000

Durch SYS 57344 (im Fall, daß Sie es bei E000 belassen) starten Sie unser OLD, und das BASIC-Programm ist wieder auferstanden.

5.5 Speicher begrenzen

Ein Nebenprodukt der Arbeit an der OLD-Routine war das Auffinden eines weiteren Vektors, der die Obergrenze des BASIC-Textes in der Bank 0 festlegt. Dadurch ist es auch beim PC128 möglich, Maschinenprogramme vor dem Überschreiben durch BASIC-Text zu schützen. Eine Rolle kann das dann spielen, wenn ein sehr langes Grafikprogramm Gefahr läuft, in die Maschinenprogrammspeicherbereiche zu gelangen. Der Vektor ist

$1212/3 dez. 4626/7 obere Speichergrenze Bank 0

Durch BANK0 : POKE4626,0 : POKE4627,224

können wir beispielsweise unser OLD-Programm schützen.

5.6 Oasen für Maschinenprogramme

Zwar verfügen wir im Commodore 128 über enorm viel Speicherplatz, und man sollte meinen, daß die Unterbringung von einigen Byte Assemblerprogramm keine Probleme bieten sollte. Weit gefehlt! Probleme treten in dem Moment auf, in dem durch das Programm Firmware-Routinen aufzurufen sind. Befindet es sich dann im RAM unter den Routinen, wird's recht kompliziert, jedesmal Bank-Umschaltungen miteinzubauen. Erfreulicherweise existiert aber ein gewaltiger Bereich, über dem kein ROM zu finden ist, so daß man sich um den ganzen Bank-Zirkus nicht zu kümmern braucht (jedenfalls nicht im Assembler-Programm). Ohne mit einem BASIC-Programmtext in Konflikte zu geraten oder mit RAM-Bereichen, die für die Grafik notwendig sind, können einige Speicherteile unterhalb von $1C00 dienlich sein.

Da bietet sich zunächst einmal der Kassettenpuffer an. Er liegt im Gebiet $0B00 bis 0BFF. Zweierlei spricht allerdings gegen das Einlagern eigener Routinen dort:

1. Es gibt mehr Datasettenbenutzer als man glaubt! Und die ärgern sich immer fürchterlich, wenn ihre Interessen übergangen werden.

2. Nach jedem RESET ist dieser Bereich gelöscht. Das ist beispielsweise für ein OLD-Programm - das ja nun gerade in Aktion treten soll - recht unangenehm.

Benutzen Sie also diesen Speicherabschnitt nur dann, wenn Sie sicher sind, daß Sie keine Datasettenoperation während der Speicherverweildauer Ihres Maschinenprogrammes brauchen, und legen Sie nur solche Programme oder Daten dort ab, bei denen Ihnen ein RESET nicht allzu weh tut.

Falls Sie am Commodore 128 ausgiebig die RS232C-Schnittstelle benutzen, dann überlesen Sie diesen Abschnitt, denn es geht um die Bereiche $0C00 bis 0CFF (RS232C-Eingabepuffer) und $0D00 bis $0DFF (das ist der Ausgabepuffer). Soweit ich feststellen konnte, wird dieser gesamte Bereich ausschließlich durch die genannte Schnittstelle benutzt, ansonsten gibt es keinen Hinderungsgrund, hier allerlei Programme oder Daten abzulegen.

Aber es kommt noch besser: Auch größere Maschinenprogramme finden Platz zwischen $1300 und 1BFF. Hier kommt ihnen nichts mehr ins Gehege. Ich empfehle Ihnen daher, das OLD-Programm aus dem Abschnitt 5.4 mittels des Monitor-Kommandos

```
T E000 E051 1300
```

dorthin zu verschieben. Weitere - noch vorzustellende - Nützlichkeiten sollen sich dort dann anschließen.

5.7 Noch einmal OLD

Gehen wir davon aus, daß OLD nun bei $1300 startet. Das Programm ist so gebaut, daß es auch verschobene BASIC-Programme wieder restaurieren kann (falls der Zeiger $2D/2E noch stimmt). Das funktioniert auch einwandfrei, wenn man die Verschiebung durch einen Grafik-Befehl bewirkt. Wenn keine Grafik initialisiert wurde, startet der BASIC-Text bei $1C00, sonst aber bei $4000. Allerdings ist es in diesem Fall wichtig, vor dem SYS-Aufruf noch die richtige Bank einzuschalten. Insgesamt heißt dann der OLD-Aufruf:

```
BANK0:SYS DEC("1300")
```

Im BASIC-Interpreter findet sich bei $4F4F der Einsprung in eine Routine, die die Zeilenlinker neu berechnet. Im C64-Modus kann man ein OLD einfach dadurch erreichen, daß man in die beiden ersten Speicherstellen des BASIC-Textes (also in den ersten Zeilenlinker) irgendwelche von Null verschiedenen Werte schreibt und dann diese Routine aufruft (sie liegt im C64-Modus an der Adresse $A533). Danach muß der Variablenstartvektor noch mit dem richtigen Wert beschrieben werden. (Ein Programm dazu finden Sie in happy computer, Ausgabe 10, (1985), S.48.)

Zwar kann man beim Commodore 128 auf diese Weise mittels $4F4F ein Programm wieder List-fähig machen, versucht man aber, eine Zeile dazuzuschreiben, bricht das System zusammen und man hat ein RESET vollführt.

5.8 LIST im Programm-Modus

Im C64-Modus führt ein LIST im Programmtext automatisch zum Abbruch des Programms. Nicht so im 128-Modus. Hier wird das Programm (oder der gewünschte Teil) brav abgebildet, und danach läuft das Programm weiter. Das ist für selbstmodifizierende Programme ein interessanter Aspekt.

5.9 CONTROL S

Erschrecken Sie bitte nicht, falls Sie unbeabsichtigt einmal gleichzeitig auf die Control- und die S-Taste gedrückt haben: Das gerade laufende Programm hält sofort an. Erst nach einem beliebigen weiteren Tastendruck setzt es die Arbeit fort. Wir haben also eine Pausentaste im Computer!

5.10 Zusätzliche Monitor-Kommandos

Die Symbole $, +, & sowie % können nicht nur - wie im Handbuch erwähnt - zum Definieren von Zahlen bestimmter Systeme (in der gleichen Reihenfolge: hexadezimal, dezimal, octal sowie binär) verwendet werden, sondern auch als Kommandos. In diesem Fall bewirken sie eine Ausgabe der gewünschten Zahl in allen Zahlensystemen. Wollen Sie beispielsweise wissen, wie die Dezimalzahl 15 in den anderen Systemen aussieht, dann geben Sie einfach ein:

```
+15
```

Es erscheint dann:

```
$0F
+15
&17
%1111
```

Eine nützliche Option, nicht wahr!

Etwas weniger klar in seiner Anwendung - und vielleicht deshalb auch im Handbuch nicht erwähnt - ist das Monitor-Kommando J (das hängt sicher mit JUMP zusammen). Soweit ich bisher herausfinden konnte, hat J die gleiche Wirkung wie G (also GO). Es startet ein Maschinenprogramm an der als Argument angegebenen Adresse. Die Syntax ist ebenfalls die gleiche wie die von G, also startet beispielsweise J $01300 ein in Bank 0 bei Adresse $1300 liegendes Maschinenprogramm.

5.11 MERGE für den C128

Außer dem OLD-Befehl vermißt mancher Benutzer des Commodore 128 noch eine Möglichkeit, BASIC-Programme zu verbinden, die durch einen MERGE-Befehl gegeben wird. Das Prinzip, das mit einem MERGE realisiert werden kann, unterstützt die strukturierte Programmierung. Es ist nämlich nun möglich, Programm-Module zu erstellen und zu speichern, die in sinnvoller Kombination zum Gesamtwerk verknüpft werden können.

Das hier abgedruckte Programm MERGE können Sie wieder direkt mit dem Monitor eingeben, wobei Sie diesmal außer dem Kommando A auch die Option zum Definieren von Speicherinhalten (das ist dann ein jeweils der Zeile vorangestelltes "größer als"-Zeichen) verwenden müssen, um die Textpassagen zu erfassen.

Programm MERGE

Programm-Module lassen sich mittels MERGE kombinieren

```
MONITOR
    PC  SR AC XR YR SP
;  FB000 00 00 00 00 F8

. 01352  A5 FA     LDA $FA
. 01354  C9 85     CMP #$85
. 01356  B0 51     BCS $13A9
. 01358  A5 2D     LDA $2D
. 0135A  85 FB     STA $FB
. 0135C  A5 2E     LDA $2E
. 0135E  85 FC     STA $FC
. 01360  38        SEC
. 01361  AD 10 12  LDA $1210
. 01364  E9 02     SBC #$02
. 01366  85 2D     STA $2D
. 01368  AD 11 12  LDA $1211
. 0136B  E9 00     SBC #$00
. 0136D  85 2E     STA $2E
. 0136F  A9 85     LDA #$85
. 01371  85 FA     STA $FA
. 01373  20 7D FF  JSR $FF7D

>01376 0D 41 4E 53 43 48 4C 55 53 53 2D 50 52 4F 47 52:
>01386 41 4D 4D 0D 45 49 4E 4C 41 44 45 4E 0D 28 5A 45:
>01396 49 4C 45 4E 4E 55 4D 4D 45 52 4E 20 4F 4B 3F 29:
>013A6 0D 00                                          :

. 013A8  60        RTS
. 013A9  A5 FB     LDA $FB
. 013AB  85 2D     STA $2D
. 013AD  A5 FC     LDA $FC
. 013AF  85 2E     STA $2E
. 013B1  A9 00     LDA #$00
. 013B3  85 FA     STA $FA
. 013B5  20 7D FF  JSR $FF7D

>013B8 0D 44 49 45 20 50 52 4F 47 52 41 4D 4D 45 20 53:
>013C8 49 4E 44 0D 4A 45 54 5A 54 20 56 45 52 42 55 4E:
>013D8 44 45 4E 0D 00                                 :

. 013DD  60        RTS
```

Das Programm besteht aus zwei Teilen, zu denen der Inhalt der Speicherstelle $FA führt. Ist dieser Inhalt gleich Null, dann wurde MERGE zum erstenmal aufgerufen. Der Vektor $2D/2E wird zwischengespeichert, vom BAISC-Textende-Vektor $1210/1 zwei abgezogen und das Ergebnis nunmehr in den Textanfangvektor $2D/E eingetragen. Schließlich schreibt das Programm eine Kennzahl (hier $85) in die Flagge $FA und meldet sich mit einem Text:

ANSCHLUSS-PROGRAMM EINLADEN (ZEILENNUMMERN OK?)

Es besteht nun die Möglichkeit, ein weiteres Modul einzuladen oder einzutippen. Außerdem können durch RENUMBER Zeilennummern hochgelegt werden, damit sie nicht in Konflikt mit dem ersten Programmteil geraten.

Ein erneuter Aufruf von MERGE verzweigt nach der Prüfung der Flagge nun in den anderen Programmteil, der lediglich den ursprünglichen Wert des Vektors $2D/E restauriert und in die Flagge wieder den Wert Null einträgt. Abschließend wird die Meldung

DIE PROGRAMME SIND JETZT VERBUNDEN

ausgegeben. Damit sind beide Programmteile nun zu einem kombiniert, sind List- und lauffähig und können durch ein erneutes RENUMBER auch von den Zeilennummern her in eine ansprechende Form gebracht werden.
Der Aufruf von MERGE geschieht durch

BANK15:SYS DEC("1352")

vorausgesetzt, daß auch Ihr MERGE dort beheimatet bleibt. MERGE kann beliebig oft angewendet werden auch dann, wenn der BASIC-Start (beispielsweise durch Einrichten eines Grafikbildschirmes) verschoben wurde.

5.12 PRIMM

Für Assembler-Programmierer interessant dürfte ein neuer Kernal-Sprungvektor sein, der im Programm MERGE verwendet wurde: PRIMM heißt er und wird mittels JSR $FF7D aufgerufen. PRIMM druckt einen Text auf den Bildschirm, der direkt nach diesem Aufruf folgt (PRint IMMediate). Die Textsequenz ist durch ein Nullbyte zu kennzeichnen. Nach dem Ausdruck wird das Programm mit dem Befehl fortgesetzt, der auf das Nullbyte folgt.

6 Der Video-Chip für den 40-Zeichen-Modus

Eine Art "Rühr mich nicht an" ist der VIC. Allerdings gibt es doch noch eine ganze Reihe von Einflußmöglichkeiten.

6.1 Der VIC ist ein VIC - ist kein VIC, ...

Jetzt sind Sie schon zu Beginn ähnlich verwirrt, wie ich es war. Lassen Sie mich daher zunächst einmal die sicheren Tatsachen erzählen, damit wir wieder Boden unter den Füßen haben:

Der VIC ist nämlich allen, die den Commodore 64 kennen, ein alter Bekannter. Dort heißt er genaugenommen MOS 6566 Video Interface Controller. Im PC 128 haben wir eine leicht geänderte Version dieses Chips vorliegen, den MOS 8564 Video Interface Controller, der sich aber - weil er auch im C64-Modus verwendet wird - allem Anschein nach nur durch 2 zusätzliche Register vom 6566 unterscheidet. Bild 6.1 zeigt Ihnen alle Register, ihre Bedeutungen und von einigen auch die üblichen Inhalte:

Register	Adressen $	dez	Bit 7	6	5	4	3	2	1	0	Normaler Inhalt Text-M.	Hires-M.	Multi color	Bemerkung
0-16	D000-D010	53248-53264	Sprite-Positionen								–	–	–	- Basic 7.0 - JRQ
17	D011	53265	msb Raster-register	veränderter Hintergrund Modus =1 normal =0	Bit-Map-Modus =1 Text =0	Bildschirm an =1 aus =0	Zeilenzahl 24=0 25=1	Smooth Scrolling Y-Verschiebung			$1B	$BB	$BB	- teilweise durch Basic 7.0 - JRQ
											abhängig von Bit 7			
18-21	D012-D015	53266-53269	LSB Rasterregister, Lichtgriffel, Sprites								–	–	–	- Basic 7.0 - JRQ
22	D016	53270	unbenutzt, beide Bits = 1		Reset 0 = VIC arbeitet	Multicolor-mode = 1	Spaltenzahl 39 = 0 40 = 1	Smooth Scrolling X-Verschiebung			$C8	$C8	$D8	- teilweise durch Basic - JRQ
23	D017	53271	Sprite-Vergrößerung in Y-Richtung								–	–	–	- Basic 7.0 - JRQ
24	D018	53272	Bildschirmspeicher-startadresse				Text: Zeichen-musterquelle Bit-Map-Mode: 0 = BitMap unten 1 = BitMap oben		unbenutzt	unbenutzt immer = 1	$15	$79	$79	- teilweise durch Basic 7.0 - JRQ - steuerbar: 2604 5
25, 26		53273-53274	VIC-Unberbrechungs-Register								–	–	–	- Basic 7.0 -JRQ
27 - 46	D01B-D02E	53275 - 53294	Farb-Register: Hintergrund, Vordergrund, Multicolor, Sprites								–	–	–	- Basic 7.0 - teilw. JRQ
47	D02F	53295	unbenutzt (alle = 1)					Tastaturkennung			$F8	$F8	$F8	?
48	D030	53296	unbenutzt (alle = 1)						Test(?)	Taktfrequenz 1 MHz = 0 2 MHz = 1	$FC	$FC	$FC	- Basic 7.0

Bild 6.1: Die VIC-Register im Commodore 128

Die beiden neuen Register sind 47 ($D02F, 53295) und 48 ($D030, 53296). 47 enthält - etwas rätselhaft - in den Bit 0 bis 2 eine Tastaturkennung. Interessanter scheint Register 48. Das Bit 0 dient dem Umschalten in den 2MHz-Betrieb, und das ist auch im 64er-Modus möglich. Allerdings ist dann auf dem 40-Zeichen-Bildschirm nichts mehr zu sehen (oder aber wüstes Geflimmer), weil der VIC diese doppelte Geschwindigkeit nicht mitmacht. Durch POKE 53296,1 kann der C64 dann alle Operationen doppelt so schnell ausführen wie sonst. Ein POKE 53296,0 läßt den Bildschirm wieder normal erscheinen und schaltet in den 1MHz-Betrieb zurück.

Nun aber kommt das Verwirrspiel: Dies ist das einzige VIC-Register, das sich auf diese Art - also einfach durch POKEs oder Ändern mittels des Monitors - verändern läßt. Alle anderen Registerumbauten äußern sich bestenfalls in einem kurzen Aufflackern des 40-Zeichen-Bildschirmes, dann ist der Einheitszustand wieder hergestellt!

Offensichtlich befinden sich die VIC-Register fest in der Hand des Unterbrechungszyklus, der eine Reihe weiterer Bestandteile des äußeren und inneren Erscheinungsbildes unseres Computers ständig steuert. Das ist aber nur im C128-Modus der Fall. Sobald Sie den C64-Modus eingeschaltet haben, kann wieder jedes Register bedient werden. Nun ist dies hier kein Werk über die Grafik des C64. Falls Sie also mehr über die Programmierung aller Register des VIC erfahren möchten, verweise ich Sie auf mein Buch "C64 Wunderland der Grafik", Markt und Technik Verlag München 1985, MT756. Das kann übrigens auch für den Assembler-Programmierer interessant sein, der es versteht, die Unterbrechungen zu beherrschen und der sich grafisch betätigen möchte.

Für den C128-Modus ist es im allgemeinen auch kaum erforderlich, VIC-Register anzusteuern, denn fast alles Erforderliche kann über das BASIC 7.0 programmiert werden. Es bleibt nur wenig übrig, was dieses kraftvolle BASIC nicht vermag:

1. Der Textmodus mit verändertem Hintergrund existiert nirgends im BASIC 7.0.
2. Die Anzahl der Zeilen und Spalten ist nicht mehr veränderbar.
3. Das "smooth scrolling" kann nicht einfach programmiert werden.
4. Der Bildschirmspeicher kann nicht verschoben werden, ebensowenig wie die Bit-Map und die Quelle der Zeichenmuster.

Wenn Sie nun nochmal Bild 6.1 ansehen, dann stellen Sie fest, daß diese Auslassungen drei Register betreffen: 17 ($D011, 53265), 22 ($D016, 53270) und 24 ($D018, 53272), von denen zweifellos das letzte das interessante ist, denn damit haben unzählige C64-Freaks eigene Zeichen benutzt, mehrere Bildschirme kreiert und Bit-Maps verschoben. Was fängt man aber mit diesem "Rühr mich nicht an"-Chip an? Assembler-Spezis werden sagen, daß man eben einfach eine eigene Unterbrechungsroutine schreibt und den Vektor $314/5 darauf richtet. Von da an können alle VIC-Spezialitäten freimütig verwendet werden. Was aber macht ein BASIC-Programmierer?

6.2 Ein Ausweg aus der Sackgasse?

Ein gründliches Durchforsten der "erweiterten Zeropage" fördert schließlich einige interessante Speicherstellen zutage. Dem Register 24 nämlich - eben gerade dem erwähnten recht interessanten! - entsprechen 2 Speicherstellen:

2604	$0A2C	VIC Text-Basis
2605	$0A2D	VIC Bitmap-Basis

Außerdem gibt es da noch eine:

2619	$0A3B	Page des Bildschirm-Speichers

(Im C64-Modus ist letztere die Entsprechung zur Speicherstelle 648.)

Sowohl 2504 als auch 2605 entsprechen - bis auf das Bit 0, das aber auch in 53272 ohne Bedeutung ist - exakt dem Register 24 im jeweiligen Modus (also Text- oder Bitmap-Modus). Alle 3 Speicherstellen werden von den Unterbrechungen nicht behelligt - jedenfalls nicht zu unserem Nachteil wie der VIC selbst. Das, was wir in diese Speicherstellen schreiben, wird flugs nach 53272 transportiert, und somit beherrschen wir den Bildschirm, die Bit-Map und die Quelle der Zeichenmuster. Oder doch nicht so ganz? Aber dazu kommen wir später noch. Sehen wir uns zunächst einmal an, was die verschiedenen Inhalte von 53272 oder 2604 oder 2605 zu bewirken imstande sind. Die Bit 4 bis 7 legen die Startadresse des Bildschirmspeichers fest. Dort findet sich im Einschaltzustand der binäre Inhalt 0001, in den Grafik-Modi aber der Inhalt 0111. Das Bild 6.2 zeigt die weiteren Möglichkeiten.

Speicherinhalt: W binär in Bits				dez.(Bits 0 bis 3 als 0 angn.)	Bildschirmstartadresse (im Abschnitt 0)		Bemerkung
7	6	5	4		dezimal	$	
0	0	0	0	0	0	0	
0	0	0	1	16	1024	400	Text-Bildschirm
0	0	1	0	32	2048	800	
0	0	1	1	48	3072	C00	
0	1	0	0	64	4096	1000	
0	1	0	1	80	5120	1400	
0	1	1	0	96	6144	1800	
0	1	1	1	112	7168	1C00	Grafik-Bildschirm
1	0	0	0	128	8192	2000	
1	0	0	1	144	9216	2400	
1	0	1	0	160	10240	2800	
1	0	1	1	176	11264	2C00	
1	1	0	0	192	12288	3000	
1	1	0	1	208	13312	3400	
1	1	1	0	224	14336	3800	
1	1	1	1	240	15360	3C00	

Bild 6.2: Die Bildschirmspeicher-Startadresse bei verschiedenen Inhalten der Bit 4 bis 7 in 53272 oder 2604 oder 2605

Erreichbar sind diese anderen Bildschirme durch die Kommandos

POKE 2604,(PEEK(2604)AND15)OR W im Text-
POKE 2605,(PEEK(2605)AND15)OR W im Grafik-Modus.

W ist dabei der Dezimalwert, der in Bild 6.2 angegeben wurde. Bevor Sie damit anfangen, alles mögliche auszuprobieren, hier noch ein paar Tips: Speichern Sie eventuell vorhandene Programme sicherheitshalber ab. Benutzen Sie - wenn Sie haben - beide Bildschirme, denn häufig verlieren Sie aus den verschiedensten Ursachen auf dem 40-Zeichen-Bildschirm jede Kontrolle über den Cursor und Befehlstexte. Davon wird dann der 80-Zeichen-Bildschirm seltener berührt, sodaß Sie relativ gute Chancen haben, ohne RESET weiterzuarbeiten. Ein letzter Tip noch: Wundern Sie sich über nichts! Die Speicherkonstruktion unseres Computers ist derart vielfältig, daß man sich nie ganz sicher sein kann, woher eigentlich manche Bildschirmdarstellungen rühren.

Sie werden bemerkt haben, daß die höchste Bildschirmstartadresse bei 15360 liegt. Pro BANK haben wir aber 64K zur Verfügung. Was können wir tun, um dieses Mißverhältnis zu beheben. Dazu müssen wir erst mal wissen, daß der VIC-Chip über nur 14 Adressenleitungen verfügt, im Gegensatz zum Prozessor, der 16 davon bedienen kann. Was bedeutet das? Die Wirkung ist, daß mit 16 Bit alle Adressen im Speicherraum zwischen

0 = 0000 0000 0000 0000 und
65535 = 1111 1111 1111 1111

ansteuerbar sind, mit 14 Bit aber liegt die Höchstgrenze bei

16383 = 11 1111 1111 1111

Das entspricht 16K, und davon befinden sich vier in einer Bank. Es gibt nun im C64-Modus ein Register im CIA2, dem sogenannten NMI-CIA (das ist der Complex Interface Adapter, also ein Ein- und Ausgabe-Baustein), welches festlegt, auf welchen 16K-Bereich in einer Bank der VIC Zugriff hat. Es handelt sich um den DATA Port A bei $DD00 (das ist dezimal 56576). Nach allen Untersuchungen liegt an derselben Stelle auch im C128-Modus dieser Chip, sodaß man das Register auch hier benutzen kann. Der VIC-Zugriff wird durch die Bit 0 und 1 gesteuert. Was man mit den verschiedenen Belegungen erreicht, zeigt Ihnen das Bild 6.3:

Speicherinhalt binär in Bits		Speicherinhalt Dezimalwert	Abschnitt	Speicherzellen (dezimal) von - bis	Bemerkung
1	0	J			
1	1	3	0	0 - 16383	Einschaltwert
1	0	2	1	16384 - 32767	
0	1	1	2	32768 - 49151	
0	0	0	3	49152 - 65535	

Bild 6.3: Der 16K-Zugriff des VIC in Zusammenhang mit dem Inhalt der Bit 0 und 1 von Port A des NMI-CIA (Speicherstelle 56576)

Im Einschaltzustand steht hierin 11, was den unteren 16K-Bereich für den VIC freigibt. Durch Kombination der Bitwerte in 2604 (oder 2605) und derer aus Bild 6.3 lassen sich nun theoretisch 64 Bildschirme definieren. Die Befehle zur Änderung des 16K-Bereiches sind:

BANK15:POKE56576,(PEEK(56576)AND252)OR I

oder

BANK15:POKE56576,(PEEK(56576)AND252)OR 3

Dabei ist I der Dezimalwert der Bitkombination in Bild 6.3 und der zweite Befehl dient zur Wiederherstellung des Normalzustandes.

Sie finden hier noch das Programm "BILDSCHIRMSP" abgedruckt, welches Ihnen das Durchprobieren sämtlicher Möglichkeiten erleichtern soll:

Programm BILDSCHIRMSP

Ein Programm zum Testen von 64 Bildschirmen

```
10 REM ***** BILDSCHIRMSPEICHER VERSCHIEBEN *****
20 INPUT"I,W";I,W:PRINTCHR$(147)
30 BANK15:POKE56576,(PEEK(56576)AND252)ORI
40 BANK0:POKE2604,(PEEK(2604)AND15)ORW
50 P=(W/16*1024+16348*(3-I))/256
60 POKE2619,P:REM (BEI I=1 UND W =0 DAZUFUEGEN:)POKEDEC("8000"),1:POKEDEC("8001"
),2
70 PRINTCHR$(147)I,W:END
80 REM ***** MOEGLICHKEIT ZUM AUSPROBIEREN, WEITER MIT CONT *****
90 REM ***** ZURUECK ZUM NORMALEN BILDSCHIRM *****
100 PRINTCHR$(147)
110 BANK15:POKE56576,199:BANK0:POKE2604,20:POKE2619,4
120 I=3:W=16:PRINTCHR$(147)I,W
130 END
```

Aber: Speichern Sie es vor dem Starten unbedingt ab, und beachten Sie die obigen Tips, denn da tauchen allerlei Merkwürdigkeiten bei der Benutzung auf.

Sollten Sie nur mit dem 40-Zeichen-Bildschirm arbeiten, empfielt es sich, alle Operationen, die mit dem neuen Bildschirm löschender- oder druckenderweise geschehen, aus dem Programm herauszunehmen. Oder aber, Sie haben die Geduld, falls etwas schiefläuft, jedesmal nach einem RESET das Programm neu zu laden. Sehen wir uns nun einige Varianten an:

I=3 und W=0 legt den Bildschirmspeicher genau auf die Speicherplätze 0 bis 999, also auf die erweiterte Zeropage. Man kann einigen Speicherstellen bei der Arbeit zusehen, beispielsweise dem Timer-Register in der Zeile 5 links. Falls Sie den 80-Zeichen-Bildschirm verwenden, können Sie dort allerlei Kommandos eingeben und zusehen, wie sich auf dem 40-Zeichen-Bildschirm Speicherstellen verändern.

I=3 und W=16 erzeugen wieder den normalen Bildschirm, wohingegen I=3 und W=32 wieder Teile der "erweiterten Zeropage" von 2048 (BASIC-Stapel) bis 3047 (Kassettenpuffer) zeigen. Auch hier gibt es wieder veränderliche Speicherstellen, wie beispielsweise 2614 (das ist ein Korrekturzähler für den 50Hz-Betrieb). Auch die beiden nächsten Kombinationen mit I=3, nämlich die mit W=48 und W=64, zeigen noch Teile des Systemspeichers. Die Kombination I=3 und W=112 sollte eigentlich den Start des BASIC-RAM ab 7168 (bis 8167) auf dem Bildschirm erscheinen lassen, was dort aber zu sehen ist, ist nicht das Programm. Eines der noch nicht gelösten Rätsel im C128!

Alle weiteren Orte des Bildschirmspeichers, W=128 bis W=240, ergeben nutzbare Bildschirme, sofern nicht die Grafik eingeschaltet wird, die sich in diesem Gebiet tummelt. Wenn wir nun statt I=3 auch die Kombinationen mit I=2,1,0 verwenden, stellen wir fest, daß hier keine normalen Bildschirme möglich sind. Weder PRINT noch Cursorbedienung laufen hier ohne Störung. Der eine oder andere Absturz kann auftreten. Bemüht man sich aber mal, mittels des Monitors in diese Speicherbereiche den Löschcode $20 einzuschreiben (mit dem F-Kommando), und poked dann den Bildschirmcode in den neuen Bildschirmspeicher hinein, dann funktioniert das einwandfrei, wenn man dabei die kritischen Speicherbereiche der I/O-Bausteine zwischen $D000 und DFFF ausläßt, und vor allem die MMU-Register bei $FF00 und folgende. Dazu kommen wir nachher noch. Jedenfalls können - mit der Einschränkung, daß wir die Zeichen durch POKE-Kommandos einschreiben müssen - 51 Bildschirme in Bank 0 definiert werden.

6.3 Eigene Zeichen benutzen

Sollten Sie mal in der Verlegenheit sein, andere als die vorhandenen Zeichen unseres Computers zu benötigen, dann ist auch das möglich. In der Speicherstelle 2604 dienen die Bit 1 bis 3 der Festlegung der Startadresse des Zeichenmusterspeichers. Bild 6.4 schlüsselt die Zuordungen auf für den ersten 16K-Bereich.

Speicherinhalt binär in Bits			Speicherinhalt dezimal(Bit 0 = 0)	Startadresse der Zeichenmuster		Bemerkung
3	2	1	Z	dezimal	$	
0	0	0	0	0	0	
0	0	1	2	2048	800	
0	1	0	4	4096	1000	Einschaltzustand
0	1	1	6	6144	1800	
1	0	0	8	8192	2000	
1	0	1	10	10240	2800	
1	1	0	12	12288	3000	
1	1	1	14	14336	3800	

Bild 6.4: Zusammenhang der Zeichenmusterquellen mit dem Inhalt der Bits 1 bis 3 in 2604

Das Umschalten auf eine andere Zeichenmusterquelle geschieht dann mittels

```
POKE 2604,(PEEK(2604)AND240)OR Z
```

Dabei ist Z die Dezimalzahl in Spalte 1 des Bildes.

Interessanterweise deutet dieses Merkmal im Einschaltzustand auf die Speicherstelle $1000 (also 4096). Wenn wir uns aber ansehen, was dort per Monitor zu finden ist, dann liegt da die erweiterte Zeropage mit allerlei wichtigen Eintragungen. Wieder eines der ungelösten Rätsel. Dieses wurde übrigens auch schon beim C64 gestellt und immer noch nicht ganz befriedigend beantwortet. Jedenfalls holt sich der Computer im Normalzustand seine Zeichenmuster aus dem Zeichen-ROM, was es nicht möglich erscheinen läßt, diese zu verändern.

Es geht aber doch:

1. Wir kopieren den Inhalt des Zeichen-ROMs in einen RAM-Bereich.
2. Wir richten den Wegweiser in den Bit 1 bis 3 der Speicherstelle 2604 auf diesen RAM-Zeichenspeicher.
3. Wir verändern die Muster, die ja nun im RAM liegen.

Das Programm "ZEICHENSATZ", hier abgedruckt, soll Ihnen das am Beispiel des Buchstaben A erläutern.

Programm ZEICHENSATZ

Der Buchstabe A erhält ein entschieden freundlicheres Gesicht

```
10 REM ***** ZEICHENSATZ UMSCHALTEN UND VERAENDERN *****
20 REM ***** HERUNTERKOPIEREN *****
30 PRINTCHR$(147)"BITTE EINIGE ZEIT GEDULD!"
40 FAST
50 BANK14
60 FORI=0 TO 4095
70 POKE DEC("3000")+I,PEEK(DEC("D000")+I)
80 NEXTI
90 REM ***** UMSCHALTEN *****
100 SLOW
110 BANK0
120 POKE2604,(PEEK(2604)AND240)OR12
130 PRINTCHR$(147)"ABRAKADABRA"
140 SLEEP2
150 REM **** VERAENDERN DES BUCHSTABEN A *****
160 B=12296
170 POKEB,102:POKEB+1,0:POKEB+2,24:POKEB+3,24
180 POKEB+4,129:POKEB+5,102:POKEB+6,60:POKEB+7,0
```

Die Zeichen verschieben wir nach $3000 und folgende. Jeweils 8 Byte gehören einem Zeichenmuster zu. Bei 12288 beginnt der "Klammeraffe" als nulltes Zeichen, der Buchstabe A hat seine 8 Bytes von 12296 bis 12303 liegen, und dort hinein poken wir andere Inhalte. Starten Sie das Programm, dann sollten Sie sich mit etwas Geduld wappnen, denn das Ko-pieren der 4096 Bytes auf diesem Weg dauert einige Weile. Vielleicht sehen Sie sich mal den Abschnitt über den programmierten Direktmodus an. Dort ist eine schnellere Methode vorgestellt. Das Programm schaltet dann die Speicherstelle 2604 auf den neuen Zeichenbereich um und druckt auf den Bildschirm das Wort ABRAKADABRA. Einen Augenblick später erscheint das A in neuem Gewand. Gefällt es Ihnen?

Bei Ihnen hat das nicht funktioniert? Dann haben Sie alles auf dem 80-Zeichen-Bildschirm laufen lassen. Die Veränderungen spielen sich aber nur auf dem 40-Zeichen-Bildschirm ab.

Interessant ist, daß die Umschaltung auf den DIN-Zeichensatz keinen Einfluß hat auf die neuen Zeichen, wenn vor dem Programmlauf schon der DIN-Zeichensatz eingeschaltet war, erscheint das dazugehörige A im neuen Look. Merkwürdig! Welchen Zeichensatz haben wir eigentlich bei $D000 herausgelesen? Wenn Sie durch gleichzeitiges Drücken der Commodore- und der Shift-Taste auf die kleinen Buchstaben umschalten, finden Sie das normale Zeichenbild. Der Grund dafür ist, daß wir in deren Zeichenmuster nicht eingegriffen haben, denn die liegen erst ab $3800. Bild 6.5 zeigt Ihnen, wo sich im Zeichen-ROM (und in gleicher Reihenfolge nach dem Kopieren ab $3000) welche Zeichen befinden:

Block	Startadresse ($)	Inhalt
0	D000	Große Buchstaben
	D200	Grafikzeichen
	D400	Große Buchstaben (Revers)
	D600	Grafikzeichen (Revers)
1	D800	Kleine Buchstaben
	DA00	Große Buchstaben, Grafikzeichen
	DC00	Kleine Buchstaben (Revers)
	DE00	Große Buchstaben, Grafikzeichen (Revers)

Bild 6.5: Die Anordnung der Zeichenmuster im Zeichen-ROM

Ein Problem ist es noch, den richtigen Ort für die neuen Zeichenmuster zu finden. Es sollte weder einer sein, der durch ein längeres BASIC-Programm überschrieben werden kann (wie im Beispiel ZEICHENSATZ), noch darf er in einem anderen 16K-Abschnitt liegen als der Bildschirmspeicher, noch sollte er bei 4096 beginnen, denn dort greift der Computer immer auf das Zeichen-ROM zu, und ab und zu braucht man die Eintragungen der erweiterten Zeropage. Ein Weg, das Problem zu lösen, ist es, den BASIC-Start

nach $4000 hochzulegen und die Zeichenmuster ab $2000 oder $3000. Das Hochlegen des Programms nach $4000 besorgt einfach der GRAPHIC-Befehl für uns. Wir brauchen dazu im Programm ZEICHENSATZ lediglich eine Zeile einzufügen:

```
25 GRAPHIC1,1:GRAPHIC0
```

Allerdings können wir dann keine Grafik-Befehle mehr verwenden, denn dann liegen die Zeichenmuster in der Bit-Map, und jede Änderung darin wirkt sich auf die Zeichen aus. Besonders verheerend wirkt ein SCNCLR1: Unsere Muster werden gelöscht, und kein Zeichen findet mehr den Weg auf den Bildschirm. Aber auch ein einfacher DRAW-Befehl kann überraschendes bewirken. Probieren Sie doch mal DRAW1,9,0TO9,199 nach dem Lauf unseres veränderten Programms. Das A hat einen kleinen Schmiß bekommen.

Übrigens: Ein Druck auf die STOP- und die RESTORE-Taste rückt alle Verhältnisse wieder ins normale Dasein zurück. Das A grinst uns dann nicht mehr an.

6.4 Mehrere Bit-Maps

Die Speicherstelle 2605 haben wir bislang sträflich vernachlässigt. Die oberen 4 Bit haben hier dieselbe Bedeutung wie in 2604, nur daß in den Grafik-Modi der Bildschirmspeicher als Farb-RAM dient, wie wir es beim COLOR-Befehl ausführlich kennenlernen. Das Bit 3 aber hat mit der Position der Bit-Map zu tun. Bild 6.6 zeigt Ihnen, was dadurch bewirkt wird (bezogen auf den unteren 16K-Bereich):

Bit 3	Bit-Map in Abschnitt 0 (dezimal) von - bis
0	0 - 7999
1	8192 - 16191

Bild 6.6: Wirkung des Bit 3 in 2605 auf die Lage der Bit-Map

Das Setzen dieses Bits 3 geschieht durch

POKE2605,PEEK(2605)OR8

das Löschen mittels

POKE2605,PEEK(2605)AND247

Probieren Sie doch mal zur Information aus, wie es ist, wenn die Bit-Map von Speicherstelle 0 bis 7999 reicht. Das können Sie durch den zweiten POKE-Befehl und ein GRAPHIC1 sehen. Allerlei los in dieser Bit-Map, nicht wahr! Aber kommen Sie nicht auf die Idee, statt GRAPHIC1 etwa GRAPHIC1,1 einzugeben. Sie löschen damit die Zeropage, und das wirkt wie das sprichwörtliche Absägen des Astes, auf dem unser Computer sitzt. Schon passiert? Ein RESET wird Ihnen aus der Verlegenheit helfen.

Nun rechnen wir rasch mal nach: Wir haben 4 Bereiche zu je 16K, auf die wir den VIC steuern können. In jeden dieser Bereiche passen 2 Bit-Maps (immer eine oben und eine unten). Das macht nach Adam Riese insgesamt 8 Bit-Maps. Dabei ist noch zu bedenken, daß der jeweilige Bildschirmspeicher im gleichen 16K-Bereich liegen muß wie die Bit-Map. Damit Sie all diese verschiedenen Möglichkeiten mal ausprobieren können, wurde das Programm "BITMAPTEST" geschrieben.

Übrigens: Mit Tabelle 11 ist unser Bild 6.3 und mit Tabelle 12 das Bild 6.2 gemeint. Wundern Sie sich bitte nicht darüber, daß alles reichlich lange dauert. Der Grund liegt darin, daß alle Grafik-Befehle auf die normale Bitmap ab $2000 und den dazugehörigen Bildschirmspeicher ab $1C00 zugeschnitten sind. Alle anderen Bit-Maps müssen - wie im C64-Modus - durch POKE-Kommandos bedient werden, was bekanntlich eines der langsamsten BASIC-Kommandos ist.

Programm BITMAPTEST

Wie viele Bit-Maps sind sinnvoll nutzbar?

```
10 REM ***** VERLAGERUNG VON BILDSCHIRM UND BIT-MAP *****
20 REM **** NEUE WERTE EINGEBEN ****
30 F=6:DEFFNA(X)=50*SIN(X/30)+100
40 PRINTCHR$(147)CHR$(17)"EINGABE DER WERTE"CHR$(17)
50 PRINTCHR$(17)"I(SIEHE TAB.11) ABSCHNITT-KENNZIFFER"CHR$(17)
60 PRINT"W(SIEHE TAB.12) BILDSCHIRM-KENNZIFFER"CHR$(17)
70 PRINT"B=0,BIT-MAP IM UNTEREN ABSCHNITT-BEREICH"
80 PRINT" =1,BIT-MAP IM OBEREN ABSCHNITT-BEREICH"CHR$(17)
90 PRINTCHR$(17):INPUT"I,W,B=";I,W,B:A1=3-I:A2=W/16*1024+A1*16384
100 A3=A1*16384+B*8192
110 P=A2/256:PRINTCHR$(17)CHR$(17)"MIT DIESEN EINGABEN HABEN SIE"
120 PRINT"DEN ABSCHNITT "A1" GEWAEHLT."
130 PRINT"IHR BILDSCHIRM STARTET BEI "A2
140 PRINT"UND IHRE BIT-MAP BEI "A3
150 PRINTCHR$(17)"IST DAS SO IN ORDNUNG?(J/N)"
160 GETA$:IFA$=""THEN160
170 IFA$="N"THEN40
180 PRINTCHR$(147)
190 REM **** SPEICHERUMBAUTEN ****
200 BANK15:POKE56576,(PEEK(56576)AND252)ORI
210 POKE56578,PEEK(56578)OR3
220 BANK0:POKE2605,(PEEK(2605)AND15)ORW
230 POKE2619,P
240 GRAPHIC1
250 POKE2605,PEEK(2605)OR(8*B)
260 FORJ=0TO7999:POKEA3+J,0:NEXTJ
270 FORJ=0TO999:POKEA2+J,F:NEXTJ
280 FORX=0TO319:Y=FNA(X)
290 BY=(XAND504)+40*(YAND248)+(YAND7):BI=7-(XAND7)
300 POKEA3+BY,PEEK(A3+BY)OR(2↑BI):NEXTX
310 GETA$:IFA$=""THEN310
320 REM **** NORMALZUSTAND WIEDERHERSTELLEN ****
330 BANK0:POKE2604,20:POKE2605,112:POKE2619,4
340 BANK15:POKE56578,63:POKE56576,199
350 GRAPHIC0:GRAPHIC5
360 PRINTCHR$(147):END

READY.
```

Nach dem RUN werden 3 Zahlen abgefragt. I entspricht dem Kennzeichen des jeweiligen 16K-Abschnittes, W dem der Bildschirmstartadresse (bezogen auf den jeweiligen 16K-Bereich) und B kann 0 oder 1 sein und legt die Bit-Map nach unten oder oben in 16K-Abschnitt. Nach der Eingabe berechnet das Programm zunächst einmal die Startadressen von Bildschirm und Bit-Map und fragt zur Sicherheit nochmal, ob diese Adressen richtig seien. Falls man sich vertan hat, kann man neue Werte einspeisen. Dann

schaltet das Programm die neue Bit-Map und den neuen Bildschirm ein, löscht die Bit-Map (bitte bringen Sie einige Minuten Geduld auf) und den Bildschirmspeicher (siehe oben) und zeichnet eine Sinuskurve (siehe oben). Ein Tastendruck stellt wieder die normalen Verhältnisse her.

Die Tests ergeben, daß 4 Bit-Maps störungsfrei arbeiten können:

16K-Abschnitt 0, oberer Bereich
: Das ist zwar die normale Bit-Map. Sie kann aber dadurch interessant werden, daß wir den Bildschirmspeicher und damit das Farb-RAM ändern können.

16K-Abschnitt 1, oberer Bereich
: Wenn der Bildschirm an das obere Ende des unteren Bereiches gelegt wird, damit er nicht dem bei $4000 beginnenden BASIC-Text in die Quere kommt, funktioniert diese Kombination einwandfrei. Der untere Bereich ist für die Bit-Map ungeeignet, weil er das BASIC-Programm beinhaltet.

16K-Abschnitt 2, beide Bereiche
: Hier sind sowohl unten als auch oben Bit-Maps möglich.

16K-Abschnitt 3
: Keiner der beiden Bereiche ist nutzbar, weil man im unteren im Ein- und Ausgabebereich herumpfuscht, im oberen aber den Speicherteil ab $FF00, der zu den Registern der MMU gehört, stört und damit unkontrollierte Verhältnisse im Computer schafft.

Sollten Sie an der Nutzung der neuen Bit-Maps Interesse haben (sie sind nützlich für bewegte Grafik: Man kann schnell zwischen fertigen Bildern hin- und herschalten oder ein Bild zeigen während, auf dem anderen Bildschirm ein neues gezeichnet wird), dann empfehle ich Ihnen ebenfalls, das Buch "C64 Wunderland der Grafik" zu lesen. Außerdem sollten Sie die wichtigsten Routinen in Assembler programmieren: BASIC ist doch reichlich langsam!

Am Ende dieses Kapitels noch ein Hoffnungsschimmer. Wir verfügen ja nicht nur über die Bank 0! Bisher haben wir so getan, als wäre dem so. Kann man denn den VIC auch dazu bringen, die Bank 1 zu bedienen? Ob das geht und eventuell wie, werden wir bei einer Besprechung der MMU feststellen.

7 Einführung in den VDC-Chip

VDC heißt 8563 Video Display Controller. Dies ist der Chip, der die Verwaltung des 80-Zeichen-Bildschirmes managt. Eine Flut von Einflußmöglichkeiten wird uns Anwendern durch die 37 Register dieses Bausteins geöffnet. Es wäre unfair, an dieser Stelle nun alle Register abzudrucken. Sie finden diese nämlich recht ausführlich kommentiert in Ihrem Handbuch (Anhang E, Seite 6 und folgende). Deshalb sollen nur die für uns angehende Grafik-Programmierer bedeutsamen herausgenommen und näher erläutert werden. Zuvor muß ich aber noch etwas über die Eigenarten dieses Chips und seine Ansprache durch uns sagen.

Weder die Register noch ein 16K-Speicherbereich, der beim GRAPHIC-Befehl schon vorgestellt wurde und den VDC mit den notwendigen Bildschirminformationen versorgt, sind in irgendeiner Bank unter den normalen Adressen (beispielsweise durch PEEK oder POKE oder mittels Monitor) zu finden. Sie führen ein vornehmes Einzeldasein fernab davon. Nur zwei Speicherstellen im VDC-Bereich zwischen $D600 und $D700 sind die Vermittler:

$D600	dient zur Angabe des gewünschten Registers,
$D601	übergibt den in das Register zu schreibenden oder den aus diesem Register gelesenen Wert.

Ein POKE-Befehl sähe also so aus:

```
BANK15:POKE DEC("D600"),Registernr.:POKE DEC("D601"),Wert
```

Und ein PEEK wäre zu realisieren durch

```
BANK15:POKE DEC("D600"),Registernr.:PRINT PEEK(DEC("D601"))
```

Das sei ja etwas kompliziert, meinen Sie? Nun ja, Sie haben recht. Und in Maschinensprache wird es noch etwas komplexer, denn die VDC-Register sollten nur dann beschrieben werden, wenn dieser Chip nicht gerade etwas anderes zu tun hat. Zwar ist er mit seiner Arbeit so schnell, daß man in BASIC kaum Störungen zu erwarten hat, in Assembler aber sind wir auch ganz schön schnell und müssen daher auf ein Status-Bit achten, welches als Bit 7 der Speicherstelle $D600 existiert. Erst wenn hier eine 1 steht, sollte $D601 angesprochen werden.

In den 16K des VDC-Speichers liegen 3 für uns interessante Bereiche, auf die wir Einfluß nehmen möchten:

1.	$0000 -	$07CF	Bildschirm-Speicher
2.	$0800 -	$0FCF	Attribut-RAM. Das entspricht etwa dem Farb-RAM.
3.	$2000 -	$3FFF	Zeichenmuster

Zum ersten Bereich fragen wir uns, wie wir dort hinein schreiben können. Außerdem ist auch hier ein Verschieben des Bildschirmes eine nette Sache. Das gleiche gilt für das Attribut-RAM. Der Zeichenmusterbereich ist für uns interessant, weil wir durch Eingriffe hier eigene Zeichen definieren können.

Schließlich ist es noch unser Anliegen, Grafik auf dem 80-Zeichen-Bildschirm zu realisieren. Auch dazu gibt es Wege. Aber gehen wir der Reihe nach vor.

Die Register 18 und 19 des VDC bestimmen die Adresse aus dem 16K-Speicher, in die ein Wert geschrieben wird. Dabei soll dann 19 das LSB und 18 das MSB der gewünschten Anschrift erhalten. Nehmen wir mal als Beispiel das Bildschirm-RAM von $0000 bis $07CF (das ist dezimal 0 bis 1999) und daraus die Adresse $0400 (also dezimal 1024). In einem Programm können wir uns die beiden Bestandteile berechnen lassen. V sei die Adresse, LO das LSB und HI das MSB davon:

```
380 HI = INT(V/256):LO = V - HI*256
```

Wir erhalten auf diese Weise für HI den Wert 4 und für LO den Wert 0. (In Hexadezimalzahlen ist das schon auf den ersten Blick klar: 04 00.) Unser Programm muß nun weiter lauten:

```
390  POKE DEC("D600"),18:POKE DEC("D601"),HI
400  POKE DEC("D600"),19:POKE DEC("D601"),LO
```

Nun muß natürlich noch der Wert, der in den so ausgewählten Speicherplatz hinein soll, übergeben werden. Zu diesem Zweck dient das VDC-Register 31. Sei der Wert allgemein ausgedrückt die Variable W, dann lautet das weitere Programm:

```
410  POKE DEC("D600"),31:POKE DEC("D601"),W
```

Es gilt nun noch zweierlei zu beachten. Es muß angegeben werden, wie viele Speicherstellen diesen Wert von der besagten Adresse an erhalten sollen. Das schreibt man als Variable N in das Register 30. Erst durch diesen Schreibvorgang wird ein Zeichenmuster oder der angegebene Wert tatsächlich in den Speicher eingetragen. Zuvor aber gibt es noch eine Kleinigkeit zu beachten: In $D600 gibt es nämlich noch ein Bit, das anzeigt, ob ein Rasterstrahl-Rücklauf stattfindet. Darauf muß gewartet werden:

```
420  WAIT DEC("D600"),32
430  POKE DEC("D600"),30:POKE DEC("D601"),N
```

Das war's eigentlich auch schon! Damit haben wir die gesamten 16K im Griff. Fassen Sie diese Programmzeilen als Unterprogramm auf, und fügen Sie einen entsprechenden Rahmen dazu, dann erhalten Sie etwas ähnliches wie unser kleines Testprogramm SCREENTEST, das hier abgedruckt steht:

Programm SCREENTEST

Erlaubt allerlei Eingriffe in das VDC-RAM

```
100 REM ******* PROGRAMM SCREENTEST *******
110 SCNCLR5:WINDOW0,12,79,24
120 REM +++ BILDSCHIRM-RAM VERAENDERN +++
130 :INPUT"ADRESSE BILDSCHIRM-RAM(0 BIS 1999)";V
140 :INPUT"GEWUENSCHTER WERT(POKE-CODE)";W
150 :INPUT"WIE OFT";N
160 :GOSUB380:SCNCLR5
170 REM +++ ATTRIBUT-RAM VERAENDERN +++
180 :PRINT"MOECHTEN SIE EIN ATTRIBUT VERAENDERN? (J/N)"
190 :GETKEYA$:IFA$="N"THEN250
200 :INPUT"ADRESSE ATTRIBUT-RAM(2048-4047)";V
210 :INPUT"GEWUENSCHTER WERT(SIEHE BESCHREIBUNG)";W
220 :INPUT"WIE OFT";N
230 :GOSUB380:SCNCLR5
240 REM +++ ZEICHENMUSTER-RAM VERAENDERN +++
250 :PRINT"SOLLEN ZEICHENMUSTER VERAENDERT WERDEN? (J/N)
260 :GETKEYA$:IFA$="N"THEN320
270 :INPUT"POKE-CODE DES ZU AENDERNDEN ZEICHENS";P:B=8192+16*P
280 :INPUT"DAS WIEVIELTE BYTE (0 BIS 7)";BY:V=B+BY
290 :INPUT"NEUER WERT";W:N=1
300 :GOSUB380:SCNCLR5
310 REM +++ WEITERE VERAENDERUNGEN +++
320 PRINT"WEITERE VERSUCHE?"
330 PRINT"(1=BILDSCHIRM / 2=ATTRIBUT / 3=ZEICHEN / 4=ENDE)"
340 GETKEYA$:IF VAL(A$)<1 OR VAL(A$)>4 THEN320
350 SCNCLR5:ON VAL(A$) GOTO 130,200,270,360
360 WINDOW0,0,79,24,1:END
370 REM ----- UNTERPROGRAMM VDC-RAM BESCHREIBEN -----
380 HI=INT(V/256):LO=V-256*HI
390 POKE DEC("D600"),18:POKE DEC("D601"),HI
400 POKE DEC("D600"),19:POKE DEC("D601"),LO
410 POKE DEC("D600"),31:POKE DEC("D601"),W
430 POKE DEC("D600"),30:POKE DEC("D601"),N
440 RETURN
```

Hier eine kurze Beschreibung: Der erste Abschnitt dient dem Einschreiben von Zeichen in den Bildschirmspeicher. Zuerst wird nach der gewünschten Adresse gefragt, dann nach dem POKE-Code des Zeichens und schließlich nach deren gewünschter Anzahl. Damit die ganzen Dialoge nicht mit der Bildschirmveränderung in Konflikt kommen, ist der Textbetrieb auf ein Fenster beschränkt worden, das die untere Bildschirmhälfte einnimmt. Es ist daher sinnvoll, Adressen zu wählen, die außerhalb des Fensters liegen. Sie können aber auch das Programm etwas verändern und dann mit 2 Bildschirmen arbeiten. Die Abfragen geschehen in diesem Fall dann auf dem 40-Zeichen-Schirm.

Der nächste Programmblock bedient das Attribut-RAM. Auch hier werden Sie zunächst nach der Adresse gefragt. Der Zusammenhang der Bildschirmspeicher- mit der Attribut-RAM-Adresse ist ähnlich wie bei der 40-Zeichen-Darstellung: Zu jeder Bildschirmzelle gibt es eine korrespondierende Attribut-RAM-Zelle. Bild 7.1 stellt die Verhältnisse dar:

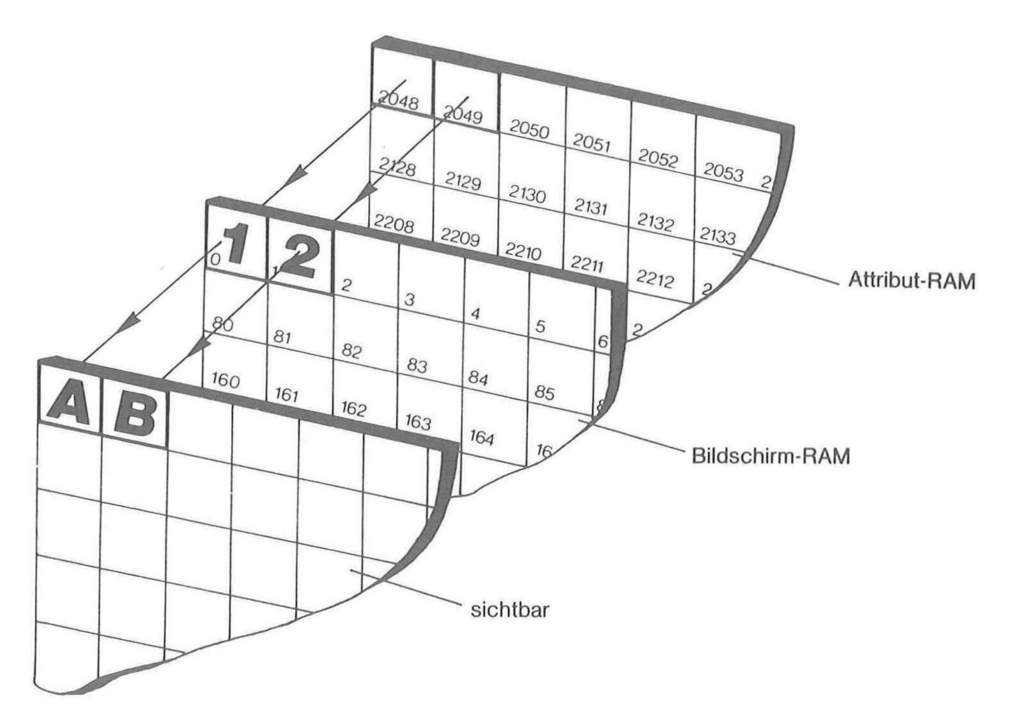


Bild 7.1: Zusammenspiel von Bildschirmspeicher und Attribut-RAM

Des weiteren werden der einzuschreibende Wert und die Anzahl der insgesamt davon betroffenen Speicherzellen erfragt - ebenso, wie es vorhin im ersten Teil geschah. Die Bedeutung der verschiedenen Einträge können Sie nachlesen im Abschnitt über die Grafik-Befehle. Falls Sie beispielsweise in Speicherstelle 80 ein A geschrieben haben, können Sie nun ein rotes, grünes oder blinkendes A (oder eines mit noch anderen Optionen) daraus machen.

Der dritte Teil unseres Programms erlaubt ein Einschreiben in den Speicherbereich, aus dem der VDC seine Zeichen abruft. Man gibt dazu den Poke-Code des zu ändernden Zeichens ein, ferner, welches der 8 Byte des Zeichenmusters man meint. Dann fragt das Programm, was statt dessen dort

hineingeschrieben werden soll. Falls Sie nun das betreffende Zeichen auf dem Bildschirm haben, erkennen Sie sofort das neue Flair. Danach haben Sie die freie Wahl, bis Sie die 4 eingeben, sie beendet das Programm. Sollten Sie wieder normale Zeichen benutzen wollen, brauchen Sie nur die ASCII/DIN-Taste zu betätigen. Dadurch wird der jeweils andere Zeichensatz ins Muster-RAM eingeladen, und unsere Mühe ist umsonst gewesen.

Behalten Sie das Programm noch eine Weile im Speicher, denn wir werden es gleich noch einmal gebrauchen für einen ganz anderen Zweck. Den ersten Teil unserer Fragen haben wir nun beantwortet. Außerdem wissen wir nun genau, wie wir auf den 16K-Speicher des VDC zugreifen können.

Sehen wir nun mal nach, inwieweit der Bildschirmspeicher verschiebbar ist. Es existieren da nämlich zwei Register im VDC-Chip, die damit zu tun haben:

Register 13	enthält das LSB,
Register 12	das MSB der neuen Bildschirmstartadresse.

Das MSB muß außerdem noch in einer Speicherstelle des normalen Speichers genannt werden, nämlich in 2606. Die Bestandteile (MSB und LSB) können wieder mittels der oben genannten Formel berechnet werden. Sie wissen ja nun auch, wie man diese Werte an die beiden Register übergibt. Also steht Ihnen nun nichts mehr im Weg, den Bildschirmspeicher innerhalb der 16K zu verschieben. Wohin kann geschoben werden? Gehen wir die 16K einmal durch, wobei wir noch daran denken sollten, daß zum einen der Bildschirm zwar nur 2000 Speicherplätze benötigt, aber unser Computer meistens 2048 Plätze behandelt (wir daher von 2048 als Platzbedarf ausgehen müssen), zum anderen sind wir nicht an pages oder ähnliche feste Startadressen gebunden wie beim VIC-Bildschirm, sondern wir können - weil wir auch das LSB festlegen - ganz frei arbeiten.

Oberhalb des normalen Bildschirm-RAMs liegt der Platz für das Attribut-RAM. Es gibt eine Möglichkeit (die wir gleich noch kennenlernen), die Farbe nicht aus diesem RAM zu holen, sondern durch ein VDC-Register bestimmen zu lassen. Zwar haben dann der Vorder- und der Hintergrund auf dem gesamten Bildschirm jeweils den gleichen Farbcode, aber das Attribut-RAM steht uns für einen weiteren Bildschirm zur Verfügung.

Ab $1000 ist bisher freier Platz, an den 2 komplette Bildschirme passen. Außerdem kann mit dem Zeichenmuster-RAM manipuliert werden. Es befinden sich beide Zeichensätze (Groß- und Kleinbuchstaben) komplett ab $2000 im RAM. Kann man auf einen der beiden verzichten - man darf

dann auch nicht die ASCII/DIN-Taste drücken -, können statt dessen noch zwei weitere Bildschirme eingerichtet werden. Insgesamt stehen uns also maximal 6 verschiedene Bildschirme zur Verfügung.

Ebenso, wie wir es mit dem Bildschirm gemacht haben, können wir auch das Attribut-RAM verschieben. Dazu dienen die folgenden Register:

Register 21	LSB der Startadresse und
Register 20	MSB

Wieder existiert im normalen RAM eine Speicherstelle, an die das MSB ebenfalls zu schreiben ist: 2607. Dazu braucht nun eigentlich nichts mehr gesagt werden, weil Sie alles schon vom Bildschirm-RAM her kennen.

Viel interessanter für uns ist das Register 25. Zwei Bit daraus, nämlich Bit 6 und Bit7, werden wir uns etwas genauer ansehen. Das Bit 6 legt fest, woher die Attribute eines Zeichens auf dem Bildschirm stammen. Im Normalfall ist dieses Bit gleich 1. Das bedeutet, daß das Attribut-RAM Quelle der Attribute ist (also Farbe, Blinken, Unterstreichen, Revers, Groß- oder Kleinschreibung). Schalten wir aber statt dessen dieses Bit auf 0, dann kommt die Farbgebung durch das Register 26 zustande. Dort dienen die Bit 0 bis 3 zum Definieren der Hintergrund-, die Bit 4 bis 7 aber der Vordergrundfarbe. Der normale Inhalt von Register 25 ist 64 (es ist also lediglich Bit 6 gesetzt), so daß wir es uns einfach machen können:

```
POKE DEC("D600"),25:POKE DEC("D601"),0
```

schaltet alle Bit auf 0, und die Farben legen wir so fest: F1=Vordergrund-, F2=Hintergrundfarbe. F = 16*F1+F2 ergibt die Farbkennzahl in Register 26. Wir setzen dann die Farben so:

```
POKE DEC("D600"),26:POKE DEC("D601"),F
```

Auf diese Weise kann auch die Farbgebung im Hochauflösungs-Modus geschehen. Das ist nämlich der Zweck des Bit 7 vom Register 25. Der Inhalt 0 an dieser Stelle sorgt für die normale Textdarstellung. Schreiben wir aber eine 1 dort hinein, dann schaltet der VDC um auf die Abbildung einer Bit-Map. Im Gegensatz zur VIC-Bit-Map benötigt die VDC-Bit-Map allerdings volle 16000 Bytes Speicherplatz, das heißt, daß das VDC-RAM praktisch vollständig als Bit-Map verwendet wird. Dafür ist aber die Auflösung in X-Richtung doppelt so groß, denn 80 Zeichen mal jeweils 8 Bit ergeben 640 verschiedene horizontale Bildpunkte. In der Vertikalen bleibt alles beim

alten: 25 Zeilen zu je 8 Bit ergeben nach wie vor 200 Bildpunkte. Insgesamt stehen uns nun 128000 verschiedene Bildschirmpositionen zur Verfügung! Das nunmehr gültige Koordinatensystem zeigt Ihnen Bild 7.2:

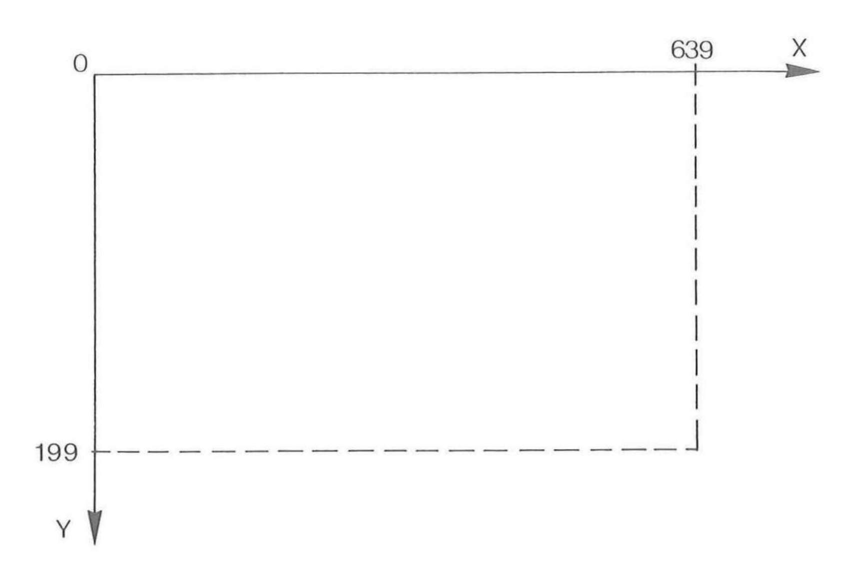


Bild 7.2: Der Hochauflösungs-Bildschirm im 80-Zeichen-Bit-Map-Modus

Schalten Sie nun einmal den Bit-Map-Modus des 80-Zeichen-Bildschirmes ein durch

```
POKE DEC("D600"),25:POKE DEC("D601"),128
```

Sie sehen, daß wir damit auch automatisch die Farben nicht mehr aus dem Attribut-RAM holen, sondern aus Register 26. Das ist ja auch viel sinnvoller, denn auch das Attribut-RAM gehört jetzt zur Bit-Map. Auf dem Bildschirm erkennen Sie in der oberen Hälfte einige gleichartig aussehende Zonen, darunter aber 4 "Türme", in denen sich helle und dunkle Bezirke abwechseln. Zur Erklärung: Der oberste Streifen ist das alte Bildschirm-RAM, darunter folgt das alte Attribut-RAM, dann ein breiter Streifen, der dem zuvor leeren Speicherbereich angehört. Die vier Türme sind das Zeichenmuster-RAM. Alles sieht etwas merkwürdig aus. Eine weitere Merkwürdigkeit werden wir nun unter Verwendung unseres vorhin benutzten Programms SCREENTEST feststellen.

Dazu sollten Sie dieses Programm umbauen, so daß der Textbetrieb auf dem 40-Zeichen-Schirm stattfinden kann (also die WINDOW-Befehle herausneh-

men und die SCNCLR5-Anweisungen). Starten Sie nach dem Umschalten in den Bit-Map-Modus das Programm, und geben Sie erst einmal 0 als Adresse ein, 255 als gewünschten Wert und etwa 70 als Anzahl. Das Ergebnis ist ein Strich am oberen Bildrand des 80-Zeichen-Schirmes. Was folgt aus diesem Ergebnis? Denken wir mal kurz nach: 70 aufeinanderfolgende Bytes der Bit-Map sind jeweils voll beschrieben worden. Hätten wir eine VIC-Bit-Map vorliegen, dann wären jetzt 8 Felder des Bildschirmes und 6/8 eines weiteren Feldes weiß geworden. Bild 7.3 zeigt dieses Resultat auf einer VIC-Bit-Map:

Byte 0	8	64
1	9	65
2	10	66
3	11	67
4	12	68
5	13	69
6	14	
7	15	

Bild 7.3: Die ersten 70 Speicherzellen der VIC-Bit-Map wurden belegt

Dort sind also jeweils 8 Bytes zu einem Feld organisiert, und es gibt 25 Reihen zu je 40 solcher Felder. Das macht die Berechnung der Bildschirmkoordinaten ziemlich kompliziert. Hier bei der VDC-Bit-Map sieht das aber anders aus. Bild 7.4 zeigt den Unterschied:

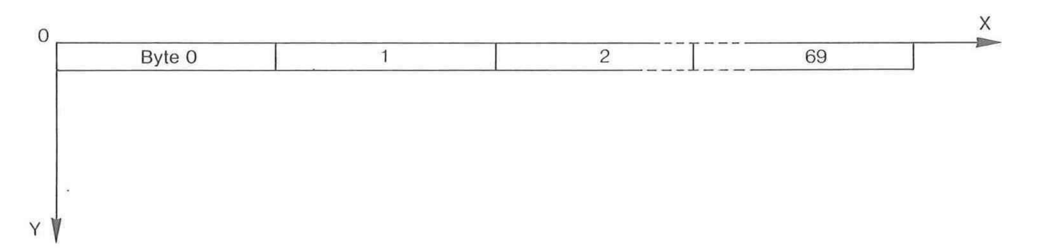


Bild 7.4: So werden die ersten 70 Speicherzellen der VDC-Bit-Map belegt

Die VDC-Bit-Map ist also nicht in solche Felder unterteilt. Vielmehr besteht sie aus 200 Reihen von je 80 Bytes. Die Koordinatenberechnung wird somit wesentlich vereinfacht.

Nun könnten wir also anfangen, ein Programm zu schreiben, welches das richtige Bit im richtigen Byte der Bit-Map aus angegebenen X- und Y-Koordinaten berechnet und dort dann, je nach Wunsch, Punkte (Bit) setzt oder löscht. Dieser Mühe glaubten sich aber Commodore 128-Besitzer zumindest im C128-Modus enthoben! Leider ist dem im Normalfall nicht so, denn alle Grafik-Befehle beziehen sich auf die VIC-Bit-Map, also den 40-Zeichen-Bildschirm.

Das hat natürlich eine Reihe gewitzter Assembler-Programmierer nicht mehr ruhig schlafen lassen. Es mußte doch einen Weg geben, die schönen Grafik-Befehle auch auf den 80-Zeichen-Bildschirm einwirken zu lassen. Tatsächlich ist das (jedenfalls, solange man keine Sprites und Shapes benötigt) auch möglich. Hauptsächlich schreitet man im Interpreter in der Punkt-Setz-Routine ein. Ein schönes - und soweit meine Tests es zeigen - auch einwandfrei funktionierendes Maschinenprogramm dazu haben kürzlich T. Rumbach und D. Winkler vorgestellt im 64er-Magazin, Ausgabe 12, 1985, Seiten 78ff.

Mit der freundlichen Genehmigung dieses Magazins ist hier das Maschinenprogramm der beiden Autoren und ein kleines Testprogramm dazu auf der Begleitdiskette enthalten. Die 80-Zeichen-Grafik wird durch BLOAD" graphik80.m" geladen und durch SYSDEC("1303") gestartet. Aus verschiedenen Gründen kann sie allerdings nur im Programm-Modus verwendet werden. Laden und Starten sollten also zu Beginn des Programms stehen.

Es gibt nun einen neuen GRAPHIC-Befehl, nämlich GRAPHIC 6,0 oder aber GRAPHIC6,1 (der zweite Parameter bestimmt, ob gelöscht wird = 1). Damit ist der VDC-Bit-Map-Modus eingerichtet. Alle folgenden Grafik-Befehle (DRAW, BOX, CIRCLE, PAINT) beziehen sich nun auf die VDC-Bit-Map. Man kann aber auch gleichzeitig die VIC-Bit-Map verwenden, wenn man jeweils durch GRAPHIC1 (oder 2,...,4) auf den 40-Zeichen-Bildschirm umschaltet. Der einzige Unterschied zu den normalen Grafik-Befehlen besteht nunmehr darin, daß X-Koordinaten bis 639 gewählt werden können. Außerdem ist nun Grafik auch im FAST-Modus sichtbar, was natürlich erhebliche Geschwindigkeitsvorteile mit sich bringt. Nähere Details sollten Sie im erwähnten Artikel der Autoren nachlesen.

8 Speicherfragen

8.1 Eine Speicherlandkarte des C128

Als Commodore-128-Benutzer stolpert man meistens zuerst einmal über den BANK-Befehl. Was es damit auf sich hat, bekommt man zwar in dem Moment heraus (oder im schlimmeren Fall nicht), wenn man mit PEEK oder POKE arbeitet und auch bei Verwendung des eingebauten Monitors, aber welche Speicher steuert man eigentlich durch die diversen Bank-Kennzahlen an, und wo befinden sich die Speicher? Das Bild 8.1 gibt zunächst einen Überblick:

Sie sehen darin unten vier große (64K)-RAM-Bereiche, von denen der untere durch BANK 0, der darüberliegende durch BANK 1 gekennzeichnet ist. Zwei weitere 64K-Bereiche können demnächst als Speichererweiterungen erworben werden. Diese vier Bereiche werden durch BANK n angesteuert, wobei n die Banknummer ist, also von 0 bis 3 reicht. Alle vier haben zwei Adressenbereiche gemeinsam: Eine "common area" (das heißt auf deutsch "gemeinsamer Bereich") von Speicherstelle 0 bis 1023, also 1K-Byte lang. Jedes Lesen oder Schreiben in diesen Teil wird automatisch in die Adressen in Bank 0 geleitet. Man braucht sich hier daher um keine Bank-Umschaltung zu kümmern. Ein schmaler Bereich zwischen $FF00 (dezimal 65280) und $FF04 (dezimal 65284) gehört zur sogenannten MMU (das heißt "memory managing unit" = "Speicherorganisations-Einheit"), um die wir uns noch kümmern werden. Dieser kleine Bereich ist deshalb allen Banks gemeinsam, weil damit die Umschaltung von einer Speicherkonfiguration zur anderen vollzogen wird, er ist gewissermaßen der Schalter dazu. Hätten wir aber eine Speicherzusammenstellung vorliegen, in der dieser Schalter nicht enthalten wäre, dann könnten wir nicht mehr zurück in andere Konfigurationen.

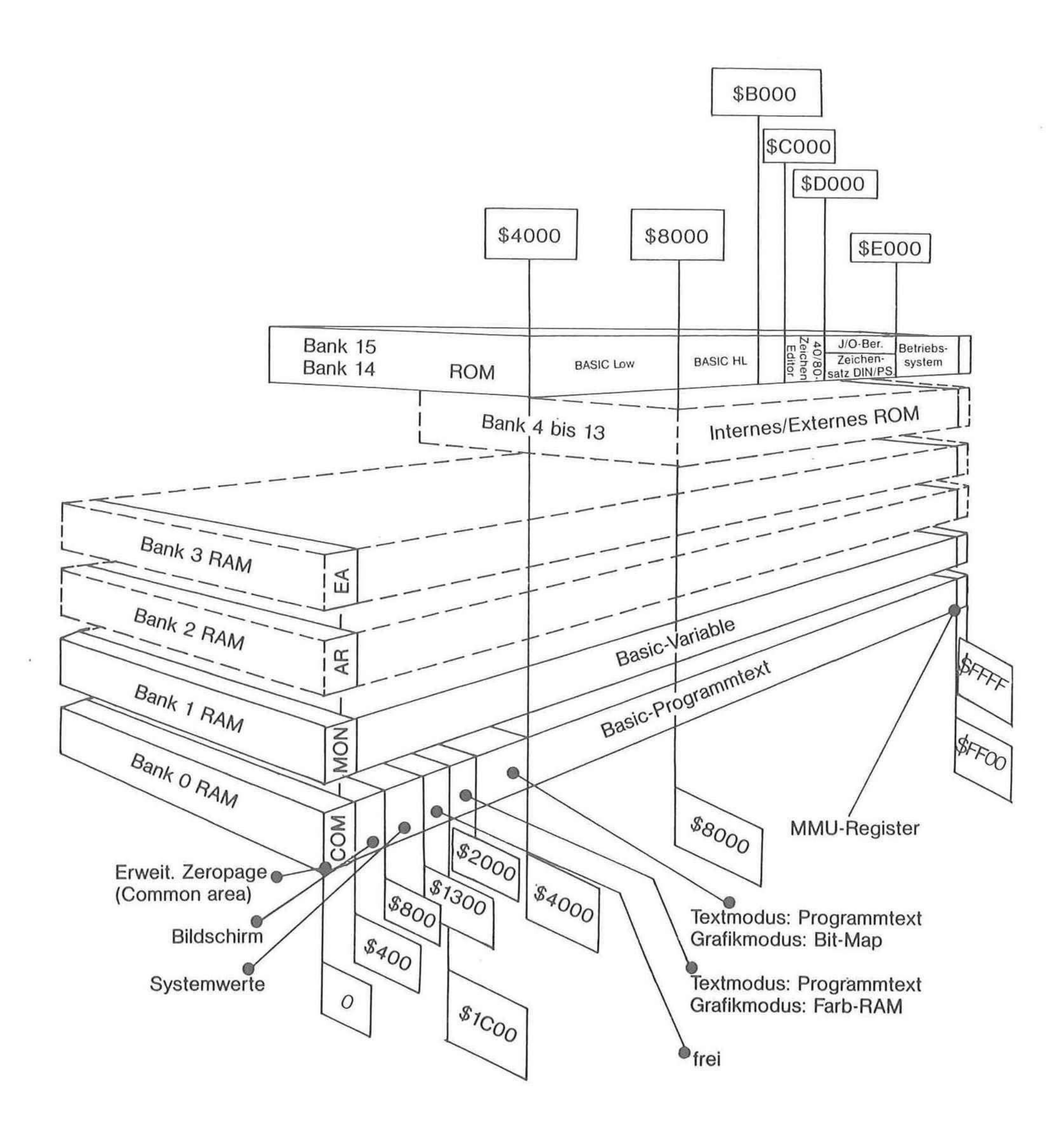


Bild 8.1: Speicherlandkarte des Commodore-128 (im C128-Modus)

Zurück zur Bank 0: Direkt oberhalb der common area befinden sich der normale Textbildschirm und die Sprite-Zeiger, die vom BASIC 7.0 benutzt werden. Von $800 bis $1300 liegt wieder Systemspeicher vor: Eine Wanderung durch die Memory-Map (ein kleiner Bezug auf die Serie von Dr. H. Hauck im 64er-Magazin) wäre hier schon ganz schön lang! Ab $1300 bis $1C00 haben wir freien Speicherplatz zur Verfügung, den man beispielsweise für Maschinenprogramme nutzen kann. Von da an wird's mehrdeutig. Falls wir nur im Textmodus arbeiten, steht uns nun der gesamte restliche Speicher (bis $FF00) für den BASIC-Programmtext zur Verfügung. Sollten aber grafische Befehle eine Rolle spielen, dann baut das Betriebssystem die Bank 0 noch etwas um: Von $1C00 bis $2000 wird ein weiteres Bildschirm-RAM eingerichtet, das als Farbspeicher dient, und zwischen $2000 und $4000 befindet sich die Bit-Map, deren Inhalt als Grafik unseren Bildschirm ziert. Erst ab $4000 ist der BASIC-Programmtext dann zu finden. Soweit also die Bank 0. Wo sind die Variablen geblieben?

Die gesamte Bank 1 ist der Tummelplatz für alle Variablen. Zwischen $400 (dort endet ja die common area) und $FF00 haben sie ihren Platz. Die Tatsache, daß sie in einer getrennten Bank liegen, hat eine Reihe von Vorteilen, aber auch Nachteile. Veränderungen des BASIC-Textes wirken sich nun nicht mehr - wie im C64-Modus - löschend auf Variable aus. Andererseits wäre es manchmal wünschenswert, auch in den freien Speicherraum oberhalb des Programms in Bank 0 Variable einzulagern, denn bei ausgiebiger Nutzung von String-Arrays hat man doch schnell die Bank 1 gefüllt. Ohne Tricks - beispielsweise mit dem Transfer-Modul aus meinem letzten Bericht - ist das aber nicht möglich.

Zu den Banks 2 und 3 kann noch nicht viel gesagt werden, denn es gibt noch keine Möglichkeiten, damit herumzuprobieren. Die im BASIC 7.0 enthaltenen Verschiebebefehle STASH, SWAP und FETCH haben mit diesen Banks zu tun. Durch die BASIC-Befehle BANK 4 bis BANK 13 kann man Einflußnahme auf die sogenannten internen und externen Funktions-ROMs ausüben. Je nach BANK-Nummer stellt man sich verschiedene Kombinationen von RAM und ROM zusammen. Auch dazu soll hier nichts weiter gesagt werden, denn diese ROMs habe ich noch nirgends gesehen.

Danach aber wird's noch einmal interessant: BANK 14 und BANK 15 blenden die Firmware in den RAM von Bank 0 ein. Der einzige Unterschied zwischen 14 und 15 liegt darin, daß man mit BANK14 den Zugriff auf das Zeichenmuster-ROM hat, wohingegen BANK 15 den Ein- und Ausgabeteil unseres Computers an die Benutzeroberfläche holt. In Bild 8.1 finden Sie den Aufbau des ganzen ROM-Paketes:

Es beginnt an der Adresse $4000 mit dem BASIC-Interpreter, dieser Teil wird als "BASIC low" bezeichnet. Sie können sich so sicher auch gleich denken, daß es ein "BASIC high" ebenfalls gibt. Das befindet sich direkt dahinter, nämlich ab $8000. Bei $B000 beginnt das Programm, das unseren Monitor am Leben erhält, bei $C000 der 40- und 80-Zeichen-Editor. Von $D000 bis $E000 existieren - wie schon vorhin erwähnt - 2 Möglichkeiten: Zeichen-ROM oder I/O-Bereich. Ab $E000 beginnt das obere ROM, das man etwas salopp auch als Betriebssystem bezeichnen könnte.

8.2 Der I/O-Bereich

Nach diesem Überblick sehen wir uns nun auch noch den Aufbau des Ein- und Ausgabebereiches an. Der ist dem im C64-Modus sehr ähnlich, was ja auch kein Wunder ist, denn er wird größtenteils auch in diesem Modus benutzt. Bild 8.2 gibt uns eine Zusammenfassung.

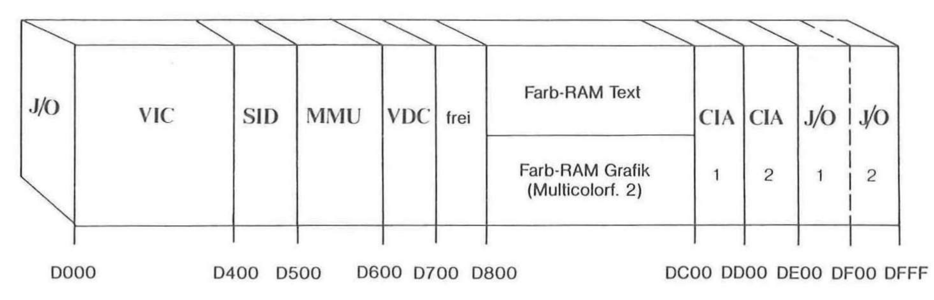


Bild 8.2: Der Ein- und Ausgabebereich im C128-Modus

Zunächst finden wir den bereits besprochenen VIC-Chip (von $D000 bis $D400). Daran schließt sich der Baustein an, der Musikliebhabern die Herzen höher schlagen läßt: der SID. Das bedeutet "sound interface device". Offenbar war zwischen $D400 und $D800 noch eine Menge Platz beim C64, denn im C128 finden sich nun zwei Bausteine, die allerhand leisten: Von $D500 bis $D600 ist die MMU ("memory managing unit") und von $D600 bis $D700 der VDC-Chip enthalten. Beide werden uns noch näher bekannt werden. Sogar ein freier Platz ist noch vorhanden. Er reicht von $D700 bis $D800. Den Rest kennen C64-Freaks noch: Es schließt sich das Farb-RAM an, das von $D800 bis $DC00 reicht. Aber Vorsicht! Ganz so einfach macht es uns der C128 nicht! Wenn Sie im Normalzustand mittels des Monitors mal dort hineinsehen, dann finden Sie zwar den Farbcode für den Text-

bildschirm, aber löschen Sie doch mal das Bit 0 der Speicherstelle $01. Sehen Sie nun nochmal ab $D800 nach: Nun finden Sie noch ein zweites Farb-RAM, nämlich das für den Code der Multicolorfarbe 2. Klammheimlich hat Commodore also hier noch weiteren Speicherbereich parat, der sonst nicht in Erscheinung tritt. Mit den Inhalten der Speicherstelle $01, die im C64-Modus eine große Rolle spielt zur Organisation der ROM-/RAM-Zusammenstellungen, werden wir uns ebenfalls gleich noch befassen.

Zwei CIAs - das kommt von "complex interface adapter" - , die den Verkehr der Peripherie mit der Zentraleinheit steuern, befinden sich zwischen $DC00 und $DD00 (das ist der CIA1, auch IRQ-CIA genannt), sowie $DD00 und $DE00 (hier haben wir den CIA2, der auch als NMI-CIA bezeichnet wird). Über deren Aufbau und Register kann man bislang nur vermuten, daß kein gravierender Unterschied zum C64 vorhanden sein wird, es gibt aber noch zu wenig Information, die genauere Aussagen ermöglicht. Zwischen $DE00 und $DF00 sowie zwischen $DF00 und dem Ende des I/O-Bereiches bei $E000 sind noch zwei Speicherabschnitte für zukünftige Erweiterungen freigehalten. Angeblich soll im letzteren optional ein DMA-Controller installiert werden.

8.3 Die Speicherstelle $01

Im C64-Modus haben die beiden Speicherstellen $00 und $01 eine besondere Rolle als sogenannter Prozessorport. Man kann durch unterschiedliche Bit-Konstellationen ROM oder RAM an verschiedenen Plätzen ein- und ausblenden. Auch im C128-Modus ist dies der Prozessorport, und wenn wir uns den Speicheraufbau ansehen, dann sollten wir auch versuchen, die Bedeutung der einzelnen Bit in diesem Modus herauszufinden. Das Ergebnis meiner Untersuchungen finden Sie zusammengefaßt in Bild 8.3:

Bit:	7	6	5	4	3	2	1	0
Bedeu-tung:	?	DIN/ASCII Umschaltg. 1 = ASCII 0 = DIN	Zusammenhang mit Kassettenport 11 = keine Taste ? 00 = Taste			Text-/Grafik-Modus 01 = Text 10 = Grafik		Steuert D800-DC00 1 = Farb-RAM für Text 0 = Farb-RAM für Grafik (Multicolorf. 2)

Bild 8.3: Der Prozessorport (Speicherstelle $01) im C128-Modus

Bit 0 ist, wie wir schon gesehen haben, verantwortlich dafür, welches Farb-RAM ab $D800 aktiv ist. Der normale Bitwert ist 1 und das Text-Farb-RAM eingeschaltet. Beim Wert 0 aber findet sich an der gleichen Stelle das Farb-RAM für die Multicolorfarbe 2.

Die Bit 1 und 2 enthalten im Grafik-Modus (also nach GRAPHIC 1...4) die Belegung 10, ansonsten aber 01 in den beiden Textmodi.

Die Bedeutung der Bit 3 bis 5 konnte noch nicht ganz geklärt werden. Anscheinend dienen sie ähnlichen Zwecken wie im C64-Modus, denn wenn eine Recordertaste gedrückt wird, wandelt sich der Inhalt der beiden Bit 4 und 5 von 11 um in 00. Im C64-Modus dienen diese Bit folgendem Zweck:

Bit 3	serielle Daten werden zum Kassettenport gesandt.
	0 = Normaler Inhalt
Bit 4	1 = keine Recordertaste gedrückt
	0 = Taste gedrückt
Bit 5	1 = Datasettenmotor aus
	0 = Motor ein

Weil aber die Recordertasten immer auch den Motor einschalten, sind beide Bit (also 4 und 5) auf 11 oder 00. Der beobachtete Normalzustand des Bits 3 im C128-Modus war ebenfalls 0.

Interessant ist Bit 6. Im Einschaltzustand findet man dort eine 1, wenn die ASCII/DIN-Taste nicht gedrückt ist. Hat man aber den DIN-Zeichensatz aktiviert, dann liegt hier eine 0 vor. Außerdem muß an dieser Stelle noch ein Nachtrag zu den Speicherfragen gemacht werden: Blicken wir nämlich in Bank 14 (oder mit dem Monitorkommando M ED000 in den Zeichenmusterspeicher, dann sehen wir zwei verschiedene Inhalte bei ASCII- und bei DIN-Betrieb. Also gibt es hier ein klammheimliches weiteres Zeichen-ROM von 4K Ausdehnung, das durch Bit 6 ein- oder ausgeblendet wird.

Die Bedeutung von Bit 7 konnte bislang noch nicht geklärt werden. Dort befindet sich eine 0.

Nur die Bit 0 und 3 können durch direkten Eingriff (also durch POKE oder Monitor-Kommandos) geändert werden. Alle anderen halten ihren Wert, bis das Ereignis eintritt (also beispielsweise der Druck auf die ASCII/DIN-Taste), das ihren Inhalt neu festlegt.

8.4 Noch mehr ROM

Da wir immer noch bei Fragen der Speicherkonfiguration sind, soll noch ein ROM-Baustein von 4K Länge erwähnt werden, der ebenfalls bei $D000 liegt. Wir werden ihn gleich erklären. Zuvor aber noch etwas Stoff zum Nachdenken:

Die Erbauer des 128 PC waren wirklich Meister im Verstecken von Speicherraum! Wie wir nachher beim Untersuchen des VDC-Chips feststellen werden, gibt es da ebenfalls nochmal 16K Speicherraum. Insgesamt hat also unser Commodore 128 in der Grundstufe folgenden Speicherplatz:

128 K	in Bank 0 und 1
60K	in Bank 15
4 K	Zeichen-ROM für ASCII
4 K	dito für DIN
1 K	Farb-RAM für Multicolor
4 K	Z80-ROM (das besprechen wir gerade)
16 K	VDC-ROM
8 K	Unteres ROM für C64-Modus
8 K	Oberes ROM für C64-Modus
----------	----------
233 K	Summa summarum!

Erstaunlich, nicht wahr: Unser COMMODORE 128 ist ein PC 233!

Nun sollen aber noch schnell diese ominösen 4K erklärt werden, die ich in der obigen Rechnung Z80-ROM genannt habe. Sicher wissen Sie, daß Sie mit dem C128 keinen reinen 6502-Mikrocomputer gekauft haben (besser gesagt 8502. Diese Zentraleinheit ist aber der 6502 so nahe, daß wir keinen Unterschied feststellen können. Sogar ein CPU-Fehler, den die 6502 aufwies, ist dort wieder eingebaut worden!). Unser Computer hat außerdem noch einen Z80-B-Mikroprozessor, der den Befehlscode des 8502 nicht versteht. Wenn aber der Computer eingeschaltet wird, versucht er zunächst einmal CP/M zu booten. Dieses Boot-Programm läuft mittels des Z80 und befindet sich im ROM, nämlich in den erwähnten 4K. Erst wenn es nichts zu booten gab - jedenfalls kein CP/M - wird der 8502 aktiv. Von diesem Augenblick an verschwindet dieses 4K-ROM wieder aus der Speicherlandschaft.

8.5 Der Memorymanager

Einen Weg, diese 233K sinnvoll zu verwalten, haben Sie ja schon kennengelernt. Aber das, was die Speicherstelle $01 da verwaltet, ist nur ein sehr kleiner Teil des Gedächtnisses unseres Computers. Den Gesamtüberblick hat die schon erwähnte MMU. Mittels insgesamt 17 Registern hat sie - und damit auch wir, wenn wir die Bedeutung der Register verstehen - alle Speichervarianten voll im Griff. Und das ist allerhand, wie Sie gleich noch sehen werden. In Bild 8.4 sind zunächst einmal alle Register mit ihren Namen und Adressen vorgestellt:

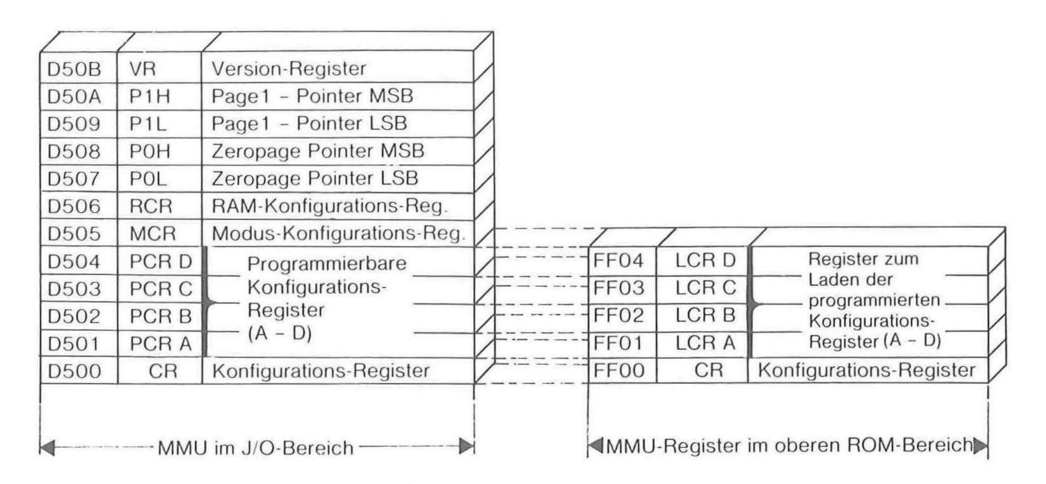


Bild 8.4: Übersicht über die MMU-Register

Sehen wir sie uns nun im einzelnen an. Da wäre erst einmal das wohl wichtigste, das CR. CR kommt von "Configuration register", und dieses Register steuert die aktuelle RAM-ROM- und I/O-Zusammenstellung.

Es befindet sich sowohl an der Speicherstelle $D500 als auch - als Doppel - bei $FF00. Falls der I/O-Bereich einmal ausgeblendet ist, haben wir noch den Schalter bei $FF00. Bild 8.5 zeigt Ihnen den Aufbau und die Bedeutung der einzelnen Bit:

Bit:	7	6	5	4	3	2	1	0
Bedeut- tung:	RAM-Bank-Auswahl 00 = Bank 0 01 = Bank 1 10 = Bank 2 11 = Bank 3		Bereich C000-FFFF 00 = ROM 01 = intern. ROM 10 = extern. ROM 11 = RAM		Bereich 8000-BFFF: 00 = Basic-ROM (high) 01 = intern. ROM 10 = extern. ROM 11 = RAM		Bereich 4000-7FFF 0 = Basic- ROM (low) 1 = RAM	Bereich D000-DFFF 0 = J/O 1 = RAM/ROM (H. Bit. 4 u. 5)

Bild 8.5: Der Aufbau des CR (Konfigurations-Register) $D500

Bit 0 steuert, was sich im Bereich $D000 bis $DFFF anfindet.

- =1 RAM/ROM-Konfiguration gemäß den Inhalten von Bit 4 und 5
- =0 I/O-Bereich ist eingeschaltet

Bit 1 richtet uns den Speicherbereich von $4000 bis $7FFF ein:

- =1 Hier befindet sich RAM
- =0 Das BASIC-low-ROM ist eingeblendet

Bit 2 und 3 kümmern sich um den Inhalt des Speicherraumes $8000 bis $BFFF.

- =00 Das BASIC-high-ROM liegt an der Benutzeroberfläche
- =01 internes ROM
- =10 externes ROM
  beide sind im für uns momentan noch ohne Interesse
- =11 RAM ist eingeschaltet

Bit 4 und 5 — Diese beiden Bit steuern den Inhalt der Speicher $C000 bis $FFFF.

=00	ROM ist eingeschaltet
=01	es liegt wieder internes
=10	oder externes ROM vor. Beides sind ebenso wie bei den Bit 2 und 3 Erweiterungen, die noch nicht aktuell sind.
=11	RAM ist eingeschaltet

Solange mit diesen beiden Bit ein ROM eingeschaltet ist, liegt immer eine Lücke im Bereich $D000 bis $DFFF. In dieser Lücke befindet sich - je nach Inhalt von Bit 1 - entweder das ganze I/O-Sortiment oder aber der Zeichenmusterspeicher.

Bit 6 und 7 — Jedesmal, wenn bei den anderen Bit eine Kombination zur Einblendung von RAM geführt hat, entscheidet nun diese letzte Paarung, welcher RAM-Bereich gemeint ist.

=00	RAM aus Bank 0
=01	RAM aus Bank 1
=10	(RAM aus Bank 2 und
=11	RAM aus Bank 3). Die beiden letzteren sind ohne Speicherergänzung noch nicht bedeutsam. Stattdessen erfolgt die Einblendung derart, daß nur Bit 6 berücksichtigt wird.

Der Einschaltwert dieses CR-Registers ist 0. Das heißt, daß alle System-ROMs aktiv sind, ebenso der I/O-Bereich. Der verbleibende Speicher gehört zu Bank 0.

Die Speicherstellen $D501 bis $D504 nennen sich PCRA bis PCRD. PCR heißt soviel wie programmierte Konfigurationsregister. Sie hängen sehr eng mit den Speicherstellen $FF01 bis $FF04 zusammen, die LCRA bis LCRD nennt. LCR kommt von "Lade das Konfigurations-Register". Der Bitaufbau dieser 8 Register ist identisch mit dem des CR, welches wir eben besprochen haben. Was haben sie für eine Wirkung, und wie benutzt man sie? Nehmen wir einmal an, daß Sie im Rahmen eines Programms vier verschiedene Speicherkonfigurationen brauchen:

Konfiguration	CR-Inhalt dazu
1) alles RAM (Bank0)	$3F (=00111111)
2) alles RAM (Bank1)	$7F (=01111111)
3) alles ROM + Bank0 + Zeichen-ROM	$01 (=00000001)
4) alles ROM + Bank1 + Zeichen-ROM	$41 (=01000001)

Nun könnten Sie natürlich jedesmal, wenn's soweit ist, den jeweiligen Code nach $FF00 schreiben. Das wäre aber überflüssige Arbeit, denn es gibt die andere Möglichkeit: Schreiben Sie zu Beginn Ihres Programms all diese verschiedenen Speichercodes in die PCR-Register, also beispielsweise den, der zur ersten Konfiguration gehört, in PCRA, den von der zweiten in PCRB und so fort. Wenn Sie nun eine bestimmte Speichergruppierung davon brauchen, dann schreiben Sie irgendeinen Wert in das dazugehörige LCR. Das ist nämlich der Trick: Sobald man beispielsweise ins LCRA irgend etwas schreibt, dann wird automatisch der Inhalt des PCRA ins CR geschrieben, was bedeutet, daß die neue Speicherkonfiguration aktiviert ist. Versucht man dagegen die LCRs zu lesen, dann erhält man statt des LCR-Inhaltes den Inhalt des dazu korrespondierenden PCR. Die oben verwendeten Beispiele stammen übrigens aus unserem Computer: Sehen Sie mal nach dem Einschalten mit dem Monitor in die LCRs hinein.

Die LCRs sind bei jeder Speicherkonfiguration ebenso erreichbar wie das CR bei $FF00.

Bisher hatten alle Register einen Verwandten im oberen Speicher-Bereich. Das ändert sich nun: Die folgenden existieren nur noch im unteren MMU-Teil. Da kommt als nächstes das MCR, also das Modus-Konfigurations-Register ins Spiel.

In Bild 8.6 sehen Sie, wie es aufgebaut ist:

Bit:	7	6	5	4	3	2	1	0
Bedeu-tung:	40/80-Zeichen-Taste 0 = gedrückt 1 = nicht ge-drückt	Betriebsart 0 = C 128 1 = C 64	EXROM 1	GAME 1	FSDIR Ausgabe für schnellen seriellen Daten-Puffer	ungenutzt 1	1	Aktive CPU: 0 = Z 80 1 = 8502

Bild 8.6: Der Aufbau des MCR (Modus-Konfigurations-Register) $D505

Bit 0 — bietet eine Möglichkeit, den Computer zum Z80-Betrieb umzustellen. Man muß dazu nur dieses Bit löschen. Allerdings wird der Computer dann abstürzen, weil er keinen gültigen Z80-Code vorfindet. Wenn man aber einen Weg dazu findet, diesen parat zu halten, dann könnte man den C128 als Z80-Computer arbeiten lassen.

Bit 1 und 2 — Diese beiden sind ungenutzt, angeblich sollen sie späteren Port-Erweiterungen dienen.

Bit 3 — hat mit dem schnellen seriellen Daten-Transport zu tun

Bit 4 und 5 — zeigen an, ob die beiden Leitungen GAME und EXROM, die mit Steckmodulen zu tun haben, belegt sind. Ist das der Fall, dann wird der C64-Modus aktiviert. C128-Module benutzen diese Leitungen nicht. Beide Leitungen können aber auch auf Ausgang (statt wie eben auf Eingang) betrieben werden. In diesem Fall haben sie mit der Verwaltung der Farb-RAM-Bank zu tun.

Bit 6 — zeigt an, welche Betriebsart aktiv ist:

=0 C128-Modus
=1 C 64-Modus

Bit 7 — schließlich gibt den Zustand der 40/80-Zeichentaste an:

=0 Taste gedrückt
=1 Taste frei

Im Abschnitt über den VIC war es schon angedeutet worden: Es gibt eine Möglichkeit in den MMU-Registern, den Zugriff des VIC auf Bank 1 zu richten. Dazu dient das RCR (RAM-Konfigurations-Register). Einen Überblick über die Möglichkeiten zeigt Bild 8.7:

Bit:	7	6	5	4	3	2	1	0
Bedeu- tung	VIC-Bankzeiger X0 = Bank 0 X1 = Bank 1 X: Bit 7 noch ohne Bedeutung		unbenutzt (geplant als Blockzeiger)		Ort der Common area: 00 = keine Common area 01 = unterer RAM-Ber. 10 = oberer RAM-Ber. 11 = ein Teil im unteren, einer im oberen RAM-Bereich		Größe der Common area 00 = 1K 01 = 4K 10 = 8K 11 = 16K	

Bild 8.7: Der Aufbau des RCR (RAM-Konfiguration-Register) $D506

Sie sehen, daß außer dem VIC-Zugriff auch die Organisation der Common areas hier geschieht.

Bit 0 und 1 — Hier wird die Größe der Common area definiert. Sie muß also keineswegs - wie im Einschaltzustand - immer 1K betragen.

=00	1K
=01	4K
=10	8K
=11	16K!

Bit 2 und 3 — Auch die Lage dieser Bereiche kann verändert werden:

=00	Keine Common area!
=01	Der gemeinsame Speicher liegt im unteren RAM-Bereich, also am Speicheranfang. Das ist der Normalzustand.
=10	In dieser Konfiguration liegt die definierte Common area am Speicherende.
=11	Der gemeinsame Speicherbereich wird aufgeteilt: Ein Teil befindet sich am Anfang, der andere am Ende des Speichers.

Bit 4 und 5 — Diese beiden Bit sind noch ungenutzt. Es wird daran gedacht, sie zu Blockzugriffen zu verwenden. Jeder Block hätte in dem Fall 256K an Speicherinhalt. Dann würden durch die 4 verschiedenen Bitkombinationen insgesamt 1 MByte RAM adressierbar werden.

Bit 6 und 7 — Da haben wir die VIC-Bankzeiger. In der vorliegenden Version des C128 spielt lediglich das Bit 6 eine Rolle:

=X0 Der VIC greift auf die Bank 0 zu

=X1 Bank 1 steht dem VIC offen! Alles, was Sie im Abschnitt über den VIC erfahren, läßt sich nun auf diese Bank übertragen. Die Anzahl der möglichen Bildschirmspeicher, Bit-Maps und Zeichen-RAMs kann verdoppelt werden.

Die Variationsbreite unseres Computers ist durch die bisher kennengelernten MMU-Register schon sehr breit. Jetzt kommt es aber noch besser. Die Register $D507 bis $D50A können nämlich die Zeropage und den Prozessorstapel verlegen! Zwar ist das vor allem für System-Programmierer interessant, aber häufig wird gerade Grafik in Assembler geschrieben. Außerdem kommt es meist besonders bei der Grafikprogrammierung auf Geschwindigkeit an. Stellen Sie sich vor, Sie könnten praktisch den gesamten Code durch immer neue Zeropages und Stapel als Zwei-Byte-Befehle schreiben!

Wie funktioniert das? Natürlich kann an dieser Stelle nur das Prinzip erklärt werden, mehr darüber sollte in Assembler-Literatur erscheinen. Übrigens haben allem Anschein nach die Hersteller der Firmware des C128 diese Möglichkeit nicht genutzt! Nun zum Prinzip. P0L (in $D507) und P0H ($D508) sind die Page-Zeiger für eine neue Zeropage. In P0L dienen die Bit 0 bis 7 zum Einschreiben der Page in einer Bank, die nun Zeropage werden soll. Die Nummer der Bank gelangt in P0H. Dafür sind dort die Bit 0 bis 3 vorgesehen. Die Bit 4 bis 7 werden nicht benutzt und sind immer 1. Im Einschaltzustand befindet sich im P0L der Wert 0 (Page 0 ist dann die Zeropage- wie sichs gehört), im P0H findet man $F0, wobei das F von den vier gesetzten oberen Bit stammt. Ansonsten ist also die Bank 0 dafür definiert. Sie merken schon, daß man auch bei dieser Umstellung zu neuen Zeropages die Common-area im Auge behalten sollte.

Ebenso, wie wir das für die Zeropage nun betrachtet haben, gilt es für den Stapel. Hier heißen die beiden Register P1L ($D509) und P1H ($D50A). Der Einschaltzustand bewirkt die Inhalte: P1L = 01 (dort befindet sich ja normalerweise der Prozessorstapel) und P1H = F0, ebenso wie beim P0H.

Eine Änderung dieser Zeiger muß zuerst die Register P0H (bzw. P1H) betreffen. Der eingeschriebene Wert wird so lange zwischengespeichert, bis auch P0L (bzw. P1L) belegt wird.

Ein letztes MMU-Register, welches aber nur gelesen werden kann, ist das VR. VR steht für Versions-Register. Die unteren 4 Byte machen eine Aussage über die vorliegende MMU-Version. In meinem C128 steht dort 0000. Die Bit 4 bis 7 geben die Anzahl der verfügbaren 64K-Banks an. Mein Computer hat dort 0010 stehen, also 2 Banks.

Die MMU ist ein faszinierendes Programmier-Instrument. Fast unübersehbar sind die Veränderungsmöglichkeiten mit all diesen Optionen. Vermutlich werden aber bald einige findige Programmierer anfangen, hier aus dem vollen zu schöpfen. Wie wäre es beispielsweise mit neuer, schnellerer Firmware, die alle diese Möglichkeiten ausnutzt?

Index